UAVCAN

Specification v1.0

Revision 2018.08.21

Overview

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

Features:

- Democratic network - no bus master, no single point of failure.
- Publish/subscribe and request/response (RPC[1]) exchange semantics.
- Efficient exchange of large data structures with automatic decomposition and reassembly.
- Lightweight, deterministic, easy to implement, and easy to validate.
- Suitable for deeply embedded, resource constrained, hard real-time systems.
- Supports dual and triply modular redundant transports.
- Supports high-precision network-wide time synchronization.
- The specification and high quality reference implementations in popular programming languages are free, open source, and available for commercial use (MIT license).

Support and feedback

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

Legal statement

UAVCAN is an interface standard open to everyone. No licensing or approval of any kind is necessary for its implementation, distribution, or use.

In no event shall the authors of the standard be liable for any damage arising, directly or indirectly, from its use.

This UAVCAN specification is distributed under the terms of the Creative Commons Attribution 4.0 International License.

[1]Remote procedure call

Table of contents

List of tables

List of figures

1 Introduction

This chapter covers the basic concepts that govern development and maintenance of the specification. The actual specification is contained in the following chapters.

1.1 Design principles

UAVCAN is designed to adhere to the following set of basic 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.

Nodes can exchange long payloads - Nodes must be provided with a simple way of exchanging large data structures that cannot fit into a single transport frame (such as GNSS solutions, 3D vectors, etc.). UAVCAN should perform automatic transfer decomposition and reassembly at the protocol level, hiding the related complexity from the application.

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

High throughput, low latency communication - UAVCAN must be suitable for use in hard real-time systems.

Simple logic, low computational requirements - UAVCAN targets a wide variety of embedded systems, from high-performance embedded on-board computers for intensive data processing (e.g., a high-performance GNU/Linux-powered machine) to extremely resource-constrained microcontrollers. The latter imposes severe restrictions on the amount of logic needed to implement the protocol.

Common high-level functions must be clearly defined - UAVCAN defines standard services and messages for common high-level functions, such as network discovery, node configuration, node software update, node status monitoring (which naturally grows into a vehicle-wide health monitoring), network-wide time synchronization, dynamic node ID allocation (a.k.a. plug-and-play node support), etc.

Open specification and reference implementations - The UAVCAN specification is open and freely available; the reference implementations are distributed under the terms of the permissive MIT License.

1.2 Capabilities

This section summarizes the capabilities of the UAVCAN protocol.

UAVCAN-based networks can accommodate up to 127 nodes on the same logical bus. More nodes can be added to the system by separating the bus into several independent logical segments interconnected via gateway nodes.

UAVCAN supports up to 32768 distinct message data types and up to 256 distinct service data types. Part of those are reserved for the standard data types defined by the specification; the rest are available for vendor- and application-specific data types. More information is provided in the chapter 5.

UAVCAN supports at least[2] eight distinct communication priority levels, defined in the section 4.1.3. Within each priority level, different types of transfers and different data types are prioritized in a well-defined deterministic manner.

The list of transport protocols supported by UAVCAN is provided in the 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 the chapter 6.

1.3 Maintenance of the standard data type set

The UAVCAN maintainers are charged with advancing the standard data type set based on the input from adopters. This feedback is gathered via the official discussion forum[3], which is open to everyone.

The set of standard data type definitions is an important part of the specification (the data structure description language (DSDL) and related concepts are described in section 3). With minor exceptions explained in the chapter 5, none of the standard data types are required to be supported by protocol implementations. Rather, their objective is to create a standardized data exchange environment allowing COTS[4] equipment manufactured by different vendors to interoperate. Non-COTS applications may avoid any dependency on the standard data type definitions (barring the aforementioned minor exceptions), relying solely on vendor-specific data types instead.

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

- A new data type can be added, possibly with a default data type ID, as long as the default data type ID doesn't conflict with one of the existing data types.
- An existing data type can be modified, as long as its version number is updated accordingly and the backward compatibility guarantees are respected.
- A major version of an existing data type can be declared deprecated.
    - Once declared deprecated, the major version will be maintained for at least two more years.
    - Deprecation will be indicated in the DSDL definition and announced via the discussion forum.
- An existing data type can be declared deprecated.
    - Once declared deprecated, the data type will be maintained for at least two more years. After this period its default data type ID may be reused for an incompatible data type.
    - Deprecation will be indicated in the DSDL definition and announced via the discussion forum.

A link to the repository containing the set of default DSDL definitions can be found on the official website[5].

1.4 Referenced sources

The UAVCAN specification contains references to the following sources:

- CiA 801 - Application note - Automatic bit rate detection.
- CiA 103 - Intrinsically safe capable physical layer.
- CiA 303 - Recommendation - Part 1: Cabling and connector pin assignment.
- IEEE 754 - Standard for binary floating-point arithmetic.
- ISO 11898-1 - Controller area network (CAN) - Part 1: Data link layer and physical signaling.
- ISO 11898-2 - Controller area network (CAN) - Part 2: High-speed medium access unit.

[2] Depending on the transport protocol.
[3] Please refer to uavcan.org.
[4] Commercial off-the-shelf equipment.
[5] uavcan.org

- 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.

2 Basic concepts

UAVCAN is a lightweight protocol designed to provide a highly reliable communication method for aerospace and robotic applications via robust vehicle bus networks. A UAVCAN network is a decentralized peer network, where each peer (node) has a unique numeric identifier - *node ID* - ranging from 1 to 127, inclusively. Nodes of a UAVCAN network can communicate using the following communication methods:

Message broadcasting - The primary method of data exchange with one-to-all publish/subscribe semantics.

Service invocation - The communication method for peer-to-peer request/response interactions[6].

For each type of communication, a predefined set of data structures is used, where each data structure has a unique identifier - the *data type ID* (DTID). Additionally, every data structure definition has a pair of major and minor semantic version numbers, which enable data type definitions to evolve in arbitrary ways while ensuring a comprehensible migration path in the event of backward-incompatible changes. Some data structures are standard and defined by the protocol specification; others may be specific to a particular application or vendor.

Since every message or service data type has its own unique data type ID, and each node in the network has its own unique node ID, a pair of data type ID and node ID can be used to support redundant nodes with identical functionality inside the same network.

Message and service data structures are defined using the Data Structure Description Language (DSDL) (chapter 3). A DSDL description can be used to automatically generate the serialization/deserialization code for every defined data structure in a particular programming language. DSDL ensures that the worst case memory footprint and computational complexity per data type are constant and easily predictable, which is paramount for hard real-time and safety-critical applications.

On top of the standard data types, UAVCAN defines a set of standard high-level functions including: node health monitoring, network discovery, time synchronization, firmware update, plug-and-play node support, and more. For more information see chapter 5.

Serialized message and service data structures are exchanged by means of the transport layer (chapter 4), which implements automatic decomposition/reassembly of long transfers into/from several transport frames[7], allowing nodes to exchange data structures of arbitrary size.

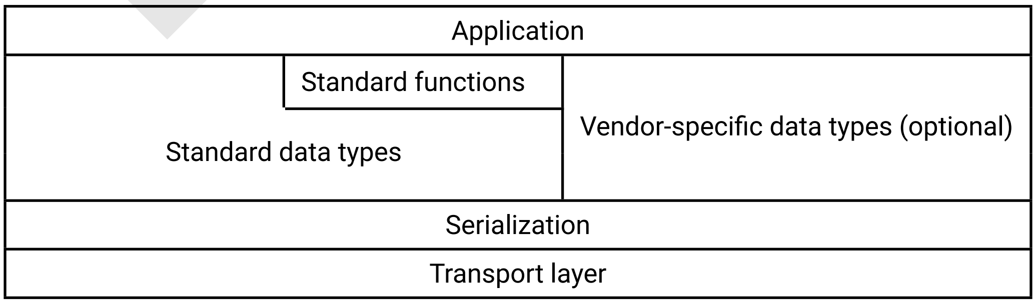


Figure 2.1: UAVCAN architectural diagram.

[6] Like remote procedure call (RPC).
[7] Here and elsewhere, a *transport frame* means a block of data that can be atomically exchanged over the network, e.g., a CAN frame.

2.1 Message broadcasting

Message broadcasting 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 broadcast message is summarized in the table 2.1.

Table 2.1: Broadcast message properties

Property	Description
Payload	The serialized message data structure.
Data type ID	Numerical identifier that indicates how the data structure should be interpreted.
Data type major version number	Semantic major version number of the data type description.
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.

2.1.1 Anonymous message broadcasting

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

UAVCAN nodes will not have an identifier initially until they are assigned one, either statically (which is generally the preferred option for applications where a high degree of determinism and high safety assurances are required) or dynamically. Anonymous messages are particularly useful for the dynamic node ID allocation feature, which is explored in detail in the 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 the chapter 4.

2.2 Service invocation

Service invocation is a two-step data exchange operation between exactly two nodes: a client and a server. The steps are[8]:

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 similar service tasks.

Information contained in service requests and responses is summarized in the table 2.2.

[8]The request/response semantic is facilitated by means of hardware (if available) or software acceptance filtering and higher-layer logic. No additional support or non-standard transport layer features are required.

Table 2.2: Service request/response properties

Property	Description
Payload	The serialized request/response data structure.
Data type ID	Numerical identifier that indicates how the data structure should be interpreted.
Data type major version number	Semantic major version number of the data type definition.
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 request and 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: data type ID, data type major version number, client node ID, server node ID, and transfer ID.

3 Data structure description language

The data structure description language (DSDL) is used to define data structures for exchange via the network. DSDL definitions are used to automatically (or manually) generate the message or service serialization/deserialization code in a particular programming language. A tool that automatically generates source code from DSDL definition files is called a *DSDL compiler*.

3.1 File hierarchy

Each DSDL definition file specifies exactly one data structure that can be used for message broadcasting, or a pair of structures that can be used for service invocation (request and response).

A DSDL source file is named using the *short data type name*, the semantic version number pair (major and minor; see the section 3.6 for more information on data type versioning), and the *default data type ID* (if needed) as shown below[9]:

```
[default DTID.]<short name>.<major version number>.<minor version number>.uavcan
```

Every defined data structure is contained in a namespace, which may in turn be *nested* within another namespace. A namespace that is not nested in another namespace is called a *root namespace*. For example, all standard data types are contained in the root namespace `uavcan`, which contains nested namespaces, such as `protocol`.

The namespace hierarchy is mapped directly to the file system directory structure, as shown in the example below:

```
uavcan/                         <- Root namespace
    equipment/                  <- Nested namespace
        ...
    protocol/                   <- Nested namespace
        341.NodeStatus.1.0.uavcan <- Data type "uavcan.protocol.NodeStatus" v1.0 with default DTID 341
        ...
    Timestamp.uavcan            <- Data type "uavcan.Timestamp", default DTID is not assigned
```

Notes:

- It is not necessary to explicitly define a default data type ID for non-standard data types (i.e., for vendor-specific or application-specific data types).
    - If the default data type ID is not defined by the DSDL definition, it will need to be assigned by the application at run time.
    - All standard data types have default data type ID values defined.
- Data type names are case sensitive, i.e., names `foo.Bar` and `foo.bar` are considered different. Names that differ only in case should be avoided, because it may cause problems on file systems that are not case-sensitive.
- Data types may contain nested data structures.
    - Some data structures may be designed for such nesting only, in which case they are not required to have a dedicated data type ID at all (neither default nor runtime-assigned).
- *Full data type name* is a unique identifier of a data type constructed from the root namespace, all nested namespaces (if any), and the short data type name, joined via the dot symbol (`.`), e.g., `uavcan.protocol.file.Read`.

[9] In this declaration, the mandatory parts are surrounded with angle brackets, and the optional parts are surrounded with square brackets.

- The total length of the full data type name must not exceed 80 characters.
- Refer to the naming rules below for the limitations imposed on the character set.

3.1.1 Service data types

Since a service invocation consists of two independent network data exchange operations, the DSDL definition for a service must define two structures:

Request part - for the request transfer (client to server).

Response part - for the response transfer (server to client).

Both request and response structures are contained within the same DSDL definition file, separated by a special statement as defined in the section 3.2.

Service invocation data structures cannot be nested into other structures.

3.2 Syntax

A data structure definition is a collection of statements. Each statement is located on a separate line. Lines are separated with the ASCII line feed character (`\n`, code 10), or with a sequence consisting of the ASCII carriage return character followed by the ASCII line feed character (`\r\n`, code 13 and 10, respectively).

The following types of statements are defined:

Attribute - used to define entities of the data type, such as data fields and constants.

Directive - directives provide instructions to the DSDL compiler.

Service response marker - separates the request and response parts of a service data type definition.

An attribute can be either of the following:

Field - a variable that can be modified by the application and exchanged via the network.

Constant - an immutable value that does not participate in network exchange.

DSDL source files may also contain human-readable comments, which are ignored by the compiler.

A message data type definition may contain the following entities:

- Attribute definitions
- Directives
- Comments

A service data type definition may contain the following entities:

- Request part attribute definitions
- Response part attribute definitions
- Request part directives
- Response part directives
- Comments

- Service response marker (always exactly one marker) (section 3.2.4)

Unless specifically stated otherwise, directives apply only to the part of the service type definition where they are defined, not crossing the boundary of the service response marker.

3.2.1 Attribute definition

Field definitions follow one of the below specified declaration patterns:

- `cast_mode field_type field_name`
- `cast_mode field_type[X] field_name`
- `cast_mode field_type[<X] field_name`
- `cast_mode field_type[<=X] field_name`
- `void_type`

Constant definitions are formed using the following declaration patterns:

- `cast_mode constant_type constant_name = constant_initializer`

Each component of the specified patterns is reviewed in detail below.

3.2.1.1 Field type

A field type declaration can be either a primitive data type (primitive data types are defined in section 3.3) or a nested data structure.

A primitive data type is referred simply by its name, e.g., `float16`, `bool`.

A nested data structure is referred by its name and the version number, separated by the ASCII dot (full stop) character. The name can be either the full name or the short name. The latter option is permitted only if the referred data type is located in the same namespace as the referring data type. The version number can be either the major version number, or both the major and the minor version numbers separated by the ASCII dot (full stop) character. In the former case, the highest available minor version number is implied. Consider the following examples, where all of the declarations refer to the same nested data type, assuming that the referring definition is located in the namespace `uavcan.protocol`:

```
NodeStatus.1
NodeStatus.1.0
uavcan.protocol.NodeStatus.1
uavcan.protocol.NodeStatus.1.0
```

A field type name can be appended with a statement in square brackets to define an array:

- Syntax `[X]` is used to define a static array of size exactly X items.
- Syntax `[<X]` is used to define a dynamic array of size from 0 to X-1 items, inclusively.
- Syntax `[<=X]` is used to define a dynamic array of size from 0 to X items, inclusively.

In the array definition statements above, `X` must be a valid integer literal according to the rules defined in the section 3.2.1.4.

Observe that the maximum size of dynamic arrays is always bounded, this ensures that the worst case memory footprint and associated computational complexity are predictable.

3.2.1.2 Field name and constant name

For a message data type, all attributes must have a unique name within the data type definition.

For a service data type, all attributes must have a unique name within the same part (request/response) of the data type definition. In other words, service type attributes can have the same name as long as they are separated by the service response marker (section 3.2.4).

Restrictions on the character set and further information are provided in the section 3.4.

3.2.1.3 *Cast mode*

Cast mode defines the rules of conversion from native values of a particular programming language to serialized field values. Cast mode may be left undefined, in which case the default will be used. Cast mode cannot be specified for nested data structures and void field types. The possible cast modes are defined below.

- `saturated` - this is the default cast mode, which will be used if the attribute definition does not specify the cast mode explicitly. For integers, it prevents an integer overflow, replacing it with saturation - for example, an attempt to write 0x44 to a 4-bit field will result in a bit field value of 0x0F. For floating point values, it prevents overflow when casting to a lower precision floating point representation - for example, 65536.0 will be converted to a `float16` as 65504.0; infinity will be preserved.
- `truncated` - for integers, discards the excessive most significant bits; for example, an attempt to write 0x44 to a 4-bit field will produce 0x04. For floating point values, overflow during downcasting will produce an infinity.

3.2.1.4 *Constant definition*

A constant must be of a primitive (section 3.3) scalar type. Arrays and nested data structures are not allowed as constant types.

A constant must be assigned with a constant initializer, which must be one of the following:

- Integer zero (0).
- Integer literal in base 10, starting with a non-zero character. E.g., `123`, `-12`.
- Integer literal in base 16 prefixed with `0x`. E.g., `0x123`, `-0x12`, `+0x123`.
- Integer literal in base 2 prefixed with `0b`. E.g., `0b1101`, `-0b101101`, `+0b101101`.
- Integer literal in base 8 prefixed with `0o`. E.g., `0o123`, `-0o777`, `+0o777`.
- Floating point literal. Fractional part with an optional exponent part, e.g., `15.75`, `1.575E1`, `1575e-2`, `-2.5e-3`, `+25E-4`. Not-a-number (NaN), positive infinity, and negative infinity are intentionally not supported in order to maximize cross-platform compatibility.
- Boolean `true` or `false`.
- Single ASCII character, ASCII escape sequence, or ASCII hex literal in single quotes. E.g., `'a'`, `'\x61'`, `'\n'`.

The DSDL compiler must convert the initializer expression to its constant type if the target type can allocate the value with no data loss. If a data loss occurs (e.g., integer overflow, floating point number decays to infinity, etc.), the DSDL compiler must refuse to compile such data type.

Note that constants do not affect the serialized data layout as they are never exchanged via the network.

3.2.1.5 *Void type*

Void type is a special field type that is intended for data alignment purposes. The specification defines 64 distinct void types as follows:

- `void1` - 1 padding bit;
- `void2` - 2 padding bits;

- ...
- `void63` - 63 padding bits;
- `void64` - 64 padding bits.

A field of a void type does not have a name and its cast mode cannot be specified. During message serialization, all void fields are filled with zero bits; during deserialization, the contents of void fields should be ignored.

3.2.2 Directives

A directive is a single case-sensitive word starting with an ASCII "at sign" character (`@`), possibly followed by space-separated arguments:

```
@directive
@directive arg1 arg2
```

All valid directives are documented in this section.

3.2.2.1 Union

Keyword: `@union`.

This directive instructs the DSDL compiler that the current message or the current part of a service data type (request or response) is a *tagged union*. A tagged union is a data structure that may encode any one of its fields at a time. Such a data structure contains one implicit field - the *union tag* - that indicates which particular field the data structure is holding at the moment. Unions are required to have at least two fields.

This directive must be placed before the first attribute definition.

3.2.2.2 Deprecated

Keyword: `@deprecated`.

This directive notifies the DSDL compiler that the current data type definition is scheduled for removal in the future. Note that it applies to the current *definition* and not to the data type as a whole, meaning that only the current version is affected, while newer versions, if available, are not to be considered deprecated unless also marked with this directive.

This directive must be placed before the first attribute definition. In the case of service data types, this directive must be placed before the first attribute definition of the request part.

The DSDL compiler can leverage this directive to amend auto-generated entities with deprecation markers appropriate to the target programming language; e.g. `[[deprecated]]` in C++, `DeprecationWarning` in Python, and so on.

As explained in the section 3.6, there are circumstances when the DSDL compiler is required to assume implicit deprecation, as if the `@deprecated` directive was provided in the definition source.

3.2.3 Comments

A DSDL description may contain comments starting from the ASCII number sign (`#`) up until the end of the current line. Comments are ignored by DSDL compilers.

3.2.4 Service response marker

A service response marker separates the request and response parts of a service data type definition. The marker consists of three ASCII minus symbols (-) in a row on a dedicated line:

```
---
```

The request part precedes the marker, and the response part follows the marker. The presence of a service response marker indicates that the current definition is a service type definition rather than a message type definition.

3.3 Primitive data types

These types are assumed to be built-in. They can be directly referenced from any data type of any namespace. The DSDL compiler should implement these types using the native types of the target programming language. An example mapping to native types is given here for C/C++.

Table 3.1: Primitive data types

Name	Bit length	Possible representation in C/C++	Value range
`bool`	1	`bool` (can be optimized for bit arrays)	$\{0, 1\}$
`intX`	$2 \leq X \leq 64$	`int8_t`, `int16_t`, `int32_t`, `int64_t`	$[-\frac{2^X}{2}, \frac{2^X}{2} - 1]$
`uintX`	$2 \leq X \leq 64$	`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`	$[0, 2^X - 1]$
`float16`	16	`float`	± 65504
`float32`	32	`float`	Approx. $\pm 10^{39}$
`float64`	64	`double`	Approx. $\pm 10^{308}$
`voidX`	$1 \leq X \leq 64$	*N/A*	*N/A*

3.4 Naming rules

3.4.1 Mandatory

Field names, constant names, and type names must contain only ASCII alphanumeric characters and underscores `[A-Za-z0-9_]`, and must begin with an ASCII alphabetic character `[A-Za-z]`. Violation of this rule must be detected by the DSDL compiler and treated as a fatal error.

3.4.2 Optional

The following rules should be checked by the DSDL compiler, but are not mandatory; their violation should not be treated as a fatal error:

• Field and namespace names should be all-lowercase words separated with underscores, and may include numbers, (e.g.: `field_name_123`).
• Constant names should be all-uppercase words separated with underscores, and may include numbers (e.g.: `CONSTANT_NAME_123`).
• Data type names should be in camel case (first letter of each word is uppercase), and may include numbers (e.g.: `TypeName123`).

3.4.3 Advisory

The following advisory rules should be considered by the data type designer:

• Message names should be nouns and/or adjectives (e.g., `BatteryStatus`); service names should

be imperatives (e.g., Restart, GetNodeInfo).
• The name of a message that carries a command should end with the word "Command"; the name of a message that carries status information should end with the word "Status".
• The name of a service that is designed to obtain or to store data should begin with the word "Get" or "Set", respectively.

3.5 Data serialization

3.5.1 General principles

A serialized data structure of type A is an ordered set of data fields joined together into a bit string according to the DSDL definition of the data structure A. The ordering of the fields follows that of the data structure definition.

Serialized bit strings do not have any implicit data entities such as padding or headers. Data type developers are advised[10] to manually align fields at byte boundaries using the void data types in order to simplify data layouts and improve the performance of serialization and deserialization routines.

Serialized fields follow the little-endian byte order[11]. One byte is assumed to contain exactly eight bits. Bits are filled from the most significant bit to the least significant bit, i.e., the most significant bit has the index 0.

Serialized data structures must be padded upon completion to one byte, with the pad bits set to zero. Lower layers of the protocol may also add additional padding as necessary[12]; however, the rules and patterns of such padding fall out of the scope of the DSDL specification.

3.5.1.1 Example

Consider the following data type definition:

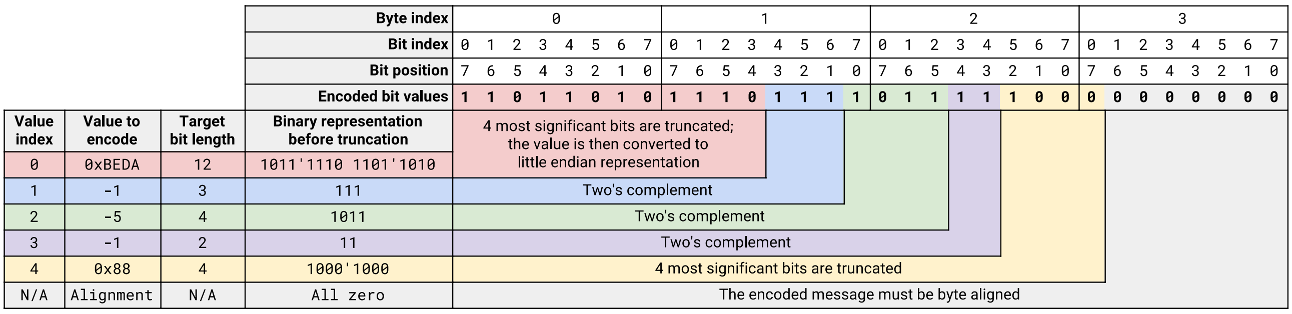
```
truncated uint12 first
truncated int3   second
truncated int4   third
truncated int2   fourth
truncated uint4  fifth
```

It can be seen that the bit layout is rather complicated because the field boundaries do not align with byte boundaries, which makes it a good case study. Suppose that we were to encode the above structure with the fields assigned the following values shown in the comments:

```
truncated uint12 first      # = 0xBEDA (48858)
truncated int3   second     # = -1
truncated int4   third      # = -5
truncated int2   fourth     # = -1
truncated uint4  fifth      # = 0x88 (136)
```

The resulting serialized byte sequence is shown on the figure 3.1.

[10] But not required.
[11] Least-significant byte (LSB) first.
[12] More on this in the chapter 4.



Byte index	0								1								2								3							
Bit index	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Bit position	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Encoded bit values	1	1	0	1	1	0	1	0	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Value index	Value to encode	Target bit length	Binary representation before truncation	
0	0xBEDA	12	1011'1110 1101'1010	4 most significant bits are truncated; the value is then converted to little endian representation
1	-1	3	111	Two's complement
2	-5	4	1011	Two's complement
3	-1	2	11	Two's complement
4	0x88	4	1000'1000	4 most significant bits are truncated
N/A	Alignment	N/A	All zero	The encoded message must be byte aligned

Figure 3.1: DSDL serialization example.

3.5.2 Scalar values

The table 3.2 lists the scalar value serialization rules.

Table 3.2: Scalar value serialization

Type	Bit length	Binary format
`bool`	1	Single bit
`int`**`X`**	X	Two's complement signed integer
`uint`**`X`**	X	Plain bits
`float16`	16	IEEE 754 binary16
`float32`	32	IEEE 754 binary32
`float64`	64	IEEE 754 binary64
`void`**`X`**	X	X zero bits; ignore when decoding

3.5.3 Nested data structures

Nested data structures are serialized directly in-place, as if their DSDL definition was pasted directly in place of their reference. No additional prefixes, suffixes, or padding is provided.

3.5.4 Fixed size arrays

Fixed-size arrays are serialized as a plain sequence of items, with each item serialized independently in place, with no alignment. No extra data is added.

Essentially, a fixed-size array of size X elements will be serialized exactly in the same way as a sequence of X fields of the same type in a row. Hence, the following two data type definitions will have identical binary representation, the only actual difference being their representation for the application if automatic code generation is used.

```
AnyType[3] array
```

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

3.5.5 Dynamic arrays

The following two array definitions are equivalent; the difference is their representation in the DSDL definition for better readability:

```
AnyType[<42] a      # Can contain from 0 to 41 elements
AnyType[<=41] b     # Can contain from 0 to 41 elements
```

A dynamic array is serialized as a sequence of serialized items prepended with an unsigned integer field representing the number of contained items - the *length field*. The bit width of the length field is a function of the maximum number of items in the array:

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

where X is the maximum number of items in the array. For example, if the maximum number of items is 251, the length field bit width must be 8 bits; if the maximum number of items is 1, the length field bit width will be just a single bit.

It is recommended to manually align dynamic arrays by prepending them with void fields so that the first element is byte-aligned, as that enables more efficient serialization and deserialization. This recommendation does not need to be followed if the size of the array elements is not a multiple of eight bits or if the array elements are of variable size themselves (e.g., a dynamic array of nested types which contain dynamic arrays themselves).

Consider the following definition:

```
void2                   # Padding - not required, provided as an example
AnyType[<42] array      # The length field is 6 bits wide (see the formula)
```

If the array contained three elements, the resulting binary representation would be equivalent to that of the following definition:

```
void2                   # Padding - not required, provided as an example
uint6 array_length      # Set to 3, because the array contains three elements
AnyType item_0
AnyType item_1
AnyType item_2
```

3.5.6 Unions

Similar to dynamic arrays, tagged unions are serialized as two subsequent entities: the union tag followed by the selected field, with no additional data.

The union tag is an unsigned integer, the bit length of which is a function of the number of fields in the union:

$$\lceil \log_2 N \rceil$$

where N is the number of fields in the union. The value serialized in the union tag is the index of the selected field. Field indexes are assigned according to the order in which they are defined in DSDL, starting from zero; i.e. the first defined field has the index 0, the second defined field has the index 1, and so on.

Constants are not affected by the union tag.

It is recommended to manually align unions when they are nested into outer data types by prepending them with void fields so that the elements are byte-aligned, as that enables more efficient serialization and deserialization.

Consider the following example:

```
@union              # In this case, the union tag requires 2 bits
uint16  FOO = 42    # A regular constant attribute
uint16  a           # Index 0
uint8   b           # Index 1
float64 c           # Index 2
uint32  BAR = 42    # Another regular constant
```

In order to encode the field b, which, according to the definition, has the data type `uint8`, the union tag should be assigned the value 1. The following structure will have an identical layout:

```
uint2 tag               # Set to 1
uint8 b                 # The actual data
```

If the value of b was 7, the resulting serialized byte sequence would be (in binary):

$$\underbrace{\texttt{01}}_{\text{tag}}\overbrace{\texttt{000001 11}}^{\text{field b}}\underbrace{\texttt{000000}}_{\text{padding}}$$

3.6 Data type compatibility and versioning

3.6.1 Rationale

As can be seen from the preceding sections, the concept of *data type* is a cornerstone feature of UAVCAN, which sets it apart from many competing solutions.

In order to be able to interoperate successfully, all nodes connected to the same bus must use compatible definitions of all employed data types. This section is dedicated to the concepts of *data type compatibility* and *data type versioning*.

A *data type* is a named set of data structures defined in DSDL. As has been explained above, in the case of message data types, the set consists of just one data structure, whereas in the case of service data types the set is a pair of request and response data structures.

Data type definitions may evolve over time as they are refined to better address the needs of their applications. In order to formalize the data type evolution process with respect to the data type compatibility concerns, UAVCAN introduces two concepts: *bit compatibility* and *semantic compatibility*, which are discussed below.

3.6.2 Bit compatibility

3.6.2.1 Definition

For the purposes of the definition that follows, a *valid serialized*[13] *representation* of a data structure A is a bit sequence that satisfies the *serialization constraints* of A.

Serialization constraints limit the set of valid bit sequences according to the data type definition. A bit sequence meets the serialization constraints if all of the following conditions are satisfied[14]:

- Each dynamic array length field contains a valid value; i.e. the value of the length field is less than the maximum number of values in the array.
- Each union tag field contains a valid value; i.e. the value of the tag field is less than the number of alternatives in the union.
- The bit sequence is sufficiently long.

A data type definition A is bit-compatible with a data type definition B if and only if the set of valid serialized representations of A is a superset of that of B.

A and B are said to be *mutually compatible* if A is compatible with B and B is compatible with A.

[13]The serialization rules are reviewed in detail in the section 3.5.

[14]Observe that serialization constraints are not affected by void-typed fields, because per the serialization rules, the values of void-typed fields are to be set to zero during serialization and to be ignored during deserialization.

3.6.2.2 Example

A *fixed-size data structure* is a structure that does not contain dynamic arrays, unions, or other structures that contain dynamic arrays or unions within themselves. As such, the bit length of an serialized representation of a fixed-size structure is constant, regardless of the data contained in the structure. Conversely, any data structure that is not fixed-size is called a *variable-size data structure*.

It stands to reason that any data structure definition is compatible with itself. The following two definitions are bit-compatible as well:

```
uint32 a
uint32 b
```

```
uint64 c
```

It should be observed that bit-compatibility is invariant to the complexity and the level of nesting of the data structure. From the above provided definitions follows that two fixed-size data structures are bit-compatible if the bit lengths of their respective serialized representations are equal.

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

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

Further, let the following be description of the data type `demo.PairVector`:

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

Then the following two definitions are bit-compatible:

```
demo.PairVector pair_vector
```

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

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

```
float16 a
float32 b
```

```
float32 a
float16 b
```

Even though the above definitions are bit-compatible, one cannot be substituted with the other. The problem of functional equivalency is addressed by the concept of semantic compatibility, explored in the section 3.6.3.

The examples above were focused on fixed-size structures. In the case of fixed-size structures, if A is compatible with B, the reverse is also true. This does not hold for variable-size structures. Consider the following example:

```
uint8[<=10] a
```

```
void1          # The maximum array length is twice lower, so the length prefix is one bit shorter.
uint16[<=5] a  # The 1-bit void field is needed to ensure identical bit offset of the array.
```

For the first definition, it is evident that since it contains one dynamic array with 11 possible length values[15], and the array type is a fixed-size structure[16], there are 11 possible bit length values for the first definition.

Likewise, for the second definition, there are 6 possible bit length values.

Since both arrays have the same bit offset from the beginning of the bit string, and taking into account the fact that the element sizes differ by an integral factor, the set of all possible serialized representations of the second definition is a subset of those of the first definition. Possible serialized representations are summarized in the table 3.3, where the columns labeled "First definition" and "Second definition" contain the number of elements in the respective arrays.

Table 3.3: Variable-size data type compatibility example

Bit length	First definition	Second definition
4	0	0
12	1	*invalid*
20	2	1
28	3	*invalid*
36	4	2
44	5	*invalid*
52	6	3
60	7	*invalid*
68	8	4
76	9	*invalid*
84	10	5

The table illustrates the fact that the first definition is compatible with the second definition, but the reverse is not true.

Complicated scenarios are possible when a bit belonging to a scalar field is handed over to a constrained field such as an array length field or a union tag field. Some interesting examples are shown in the table 3.4, together with a set of valid serialized representation patterns. Remember that the bits belonging to void-typed fields are ignored during deserialization.

[15]Which are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.
[16]Built-in types can be considered a special case of fixed-size data structures.

Table 3.4: Complex bit compatibility examples

	A	B	C	D	E

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

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

Compatible with	B	A	A, B, D	A, B, C	*(none)*

3.6.3 Semantic compatibility

3.6.3.1 Definition

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

3.6.3.2 Example

Despite using different binary layouts, the following two definitions are semantically compatible and also bit-compatible:

```
uint16 FLAG_A = 1
uint16 FLAG_B = 256
uint16 flags
```

```
uint8 FLAG_A = 1
uint8 FLAG_B = 1
uint8 flags_a
uint8 flags_b
```

Therefore, the definitions can be used interchangeably[17].

3.6.4 Data type versioning

3.6.4.1 Versioning principles

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

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

[17]It should be noted here that due to different set of fields and constants, the source code auto-generated from the provided definitions may be not drop-in replaceable, requiring changes in the application. However, application compatibility is orthogonal to data type compatibility.

In order to ensure a deterministic application behavior and ensure a robust migration path as data type definitions evolve, UAVCAN requires that all data type definitions that share the same major version number greater than zero must be semantically compatible with each other and mutually bit-compatible with each other.

Observe that the data type name or its ID do not affect its compatibility. Regardless, the default data type ID and/or the name of a data type should not be changed after its release, as that would essentially construe the release of a new data type.

In order to ensure predictable and repeatable behavior of applications that leverage UAVCAN, the standard requires that once a data type definition is released, it cannot undergo any modifications to its attributes or directives anymore. Essentially, released data type definitions are to be considered immutable excepting comments and whitespace formatting.

Therefore, substantial modifications of released data types are only possible by releasing new definitions of the same data type. If it is desired and possible to keep the same major version number for a new definition of the data type, the minor version number of the new definition shall be one greater than the newest existing minor version number before the new definition is introduced. Otherwise, the major version number shall be incremented by one and the minor version shall be set to zero.

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

3.6.4.2 *Major version release constraints*

The DSDL specification limits the number of coexisting major data type versions in order to simplify support of the data type versioning system at the transport layer and simplify the management of legacy data type definitions. As such, at any given moment, the difference between the highest released major version number and the lowest released major version number of any given data type must not exceed 3.

For example, the following set of released data type definition versions is valid and permissible: {0.1, 0.2, 0.3, 1.0, 1.1, 2.0, 2.1, 2.2, 3.0}, because the difference between the newest released major version (3) and the oldest released major version (0) does not exceed 3. The set of the minor versions is not subjected to any constraints, and as such, there are no limits on the set of concurrently released minor versions.

Continuing with the above example, if it were necessary to release a newer data type definition under a new major version of 4, the oldest major version of 0 would have to be removed first. Otherwise, the maximum major version number difference constraint would be violated. Observe that the actual number of published major versions is irrelevant; the constraint only applies to the difference between the highest and the lowest released major versions. For example, shall the version 2 be deprecated and removed while the versions 0 and 1 were still around, the requirement to remove the version 0 before publishing the version 4 would still hold. The resulting set of versions may then look like this: {1.0, 1.1, 3.0, 4.0}.

If the difference between the highest and the lowest available major version numbers exceeds 2, the DSDL compiler must assume that the oldest available definition is marked with an implicit `@deprecated` directive (section 3.2.2), even if it is not explicitly provided in the definition.

If the difference between the highest and the lowest available major version numbers exceeds 3, the DSDL compiler must refuse to process the data type and abort with an error.

3.6.4.3 *Data type version selection*

There are two aspects to the problem of data type version selection: compile-time behavior and runtime behavior. They are explored in this section.

As far as compile-time data type version selection is concerned, the DSDL compiler is required to compile every available major data type version separately, allowing the application to choose any available major version at runtime. However, there may be more than one minor version available per major version; the DSDL compiler must resolve this ambiguity by always selecting the newest available minor version per major version at the time of compilation.

For example, consider the following set of data type definition versions: {0.1, 0.2, 0.3, 1.0, 1.1, 2.0, 2.1, 2.2, 3.0}. As there are four different major data type versions (0, 1, 2, and 3), the DSDL compiler will make four independent definitions available for the application. Following the principle of choosing the newest available minor version, the resulting set of definitions available at runtime will be as follows: {0.3, 1.1, 2.2, 3.0}.

Seeing as the minor version ambiguity is resolved statically, this information becomes irrelevant for the protocol at runtime. While implementations can keep the minor version information for diagnostic purposes, it is completely unnecessary at the transport layer. As such, the transport layer (which is specified in the chapter 4) does not concern itself with the minor data type version information, whereas the major data type version is attached to every transfer.

The implication is that upon reception of a transfer, the node will use the appropriate data type definition according to the major data type version information attached to the transfer; whereas the minor versions used by the emitter and the receiver may mismatch. The possibility of a minor version mismatch is acceptable because, by definition, all data type definitions sharing the same major version number are mutually semantically compatible.

When initiating a data exchange (e.g. broadcasting a message or invoking a service), the node is free to choose the major data type version freely, according to its own application logic. Nodes that provide services (i.e., servers) must respond to requests using the same major service data type version that was used in the request. Again, the minor version number may mismatch, but by the compatibility requirement this is acceptable.

3.6.4.4 *Versioning example*

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

```
# sirius_cyber_corp.golgafrincham_b_ark.cryopod.Status.0.1

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

# Status flags in the low byte
uint16 FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint16 FLAG_COOLING_SYSTEM_B_ACTIVE = 2
# Error flags in the high byte
uint16 FLAG_PSU_MALFUNCTION = 8192
uint16 FLAG_OVERHEATING     = 16384
```

```
uint16 FLAG_CRYOBOX_BREACH   = 32768
# Storage for the above defined flags
uint16 flags
```

The definition has been deployed to the first prototype for initial lab tests. Since the definition was experimental, the major version number was set to zero, to signify the tentative nature of the definition. Suppose that upon completion of the first trials it was identified that the units must track their power consumption in real time, for each of the three redundant power supplies independently. The definition has been amended appropriately.

It is easy to see that the amended definition shown below is neither semantically compatible nor bit-compatible with the original definition; however, it shares the same major version number of zero, because the backward compatibility rules do not apply to zero-versioned data types to allow for low-overhead experimentation before the system is fully deployed and fielded.

```
# sirius_cyber_corp.golgafrincham_b_ark.cryopod.Status.0.2

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

saturated float32 power_consumption_0     # Power consumption by the redundant PSU 0 [watt]
saturated float32 power_consumption_1     # likewise for PSU 1
saturated float32 power_consumption_2     # likewise for PSU 2

# Status flags in the low byte
uint16 FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint16 FLAG_COOLING_SYSTEM_B_ACTIVE = 2
# Error flags in the high byte
uint16 FLAG_PSU_MALFUNCTION = 8192
uint16 FLAG_OVERHEATING      = 16384
uint16 FLAG_CRYOBOX_BREACH   = 32768
# Storage for the above defined flags
uint16 flags
```

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

Having collected empirical data from the fielded systems, the Sirius Cybernetics Corporation has identified a shortcoming in the v1.0 definition, which was corrected in an updated definition. Since the updated definition, which is shown below, is mutually semantically compatible[18] with v1.0, the major version number was kept the same and the minor version number was incremented by one:

```
# sirius_cyber_corp.golgafrincham_b_ark.cryopod.Status.1.1

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

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

# Status flags
uint8 STATUS_FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint8 STATUS_FLAG_COOLING_SYSTEM_B_ACTIVE = 2
uint8 status_flags

# Error flags
uint8 ERROR_FLAG_PSU_MALFUNCTION = 5
```

[18]The topic of data serialization is explored in detail in the section 3.5.

```
uint8 ERROR_FLAG_OVERHEATING     = 6
uint8 ERROR_FLAG_CRYOBOX_BREACH  = 7
uint8 error_flags
```

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

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

```
# sirius_cyber_corp.golgafrincham_b_ark.cryopod.Status.2.0

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

float32[3] power_consumption    # Power consumption by the PSU

# Status flags
uint8 STATUS_FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint8 STATUS_FLAG_COOLING_SYSTEM_B_ACTIVE = 2
uint8 status_flags

# Error flags
uint8 ERROR_FLAG_PSU_MALFUNCTION = 5
uint8 ERROR_FLAG_OVERHEATING     = 6
uint8 ERROR_FLAG_CRYOBOX_BREACH  = 7
uint8 error_flags
```

Now, nodes using v1.0, v1.1, and v2.0 definitions can still coexist on the same network, but they are not guaranteed to understand each other unless they support all of the used data type definitions.

In practice, nodes that need to maximize their compatibility are likely to employ all existing major versions of each used data type. If there are more than one minor versions available, the highest minor version within the major version should be used, to take advantage of the latest changes in the data type definition. It is also expected that in certain scenarios some nodes may resort to publishing the same message type using different major versions concurrently to circumvent compatibility issues (in the example reviewed here that would be v1.1 and v2.0).

3.7 Data type ID

Whenever a data structure is transferred over the bus, it is accompanied by a non-negative integer - a *data type ID*. The data type ID (together with the major version number, which is also exchanged over the bus together with the data type ID) is used by receiving nodes to determine which data type definition to use to process the received data structure. The data type ID value does not affect data type compatibility.

It stands to reason that in order to be able to interoperate successfully, every node connected to the bus must use identical mapping between data types and their identifiers.

There are two *kinds* of data types: message data types and service data types. Each kind has an independent set of data type identifiers. Each kind has a reserved subset which is used for standard data type definitions, and a dedicated subset for vendor-specific data type definitions. More info on the reserved subsets is provided in the chapter 5. The permitted ranges of data type ID values are specified in the table 3.5.

Table 3.5: Data type ID ranges per data kind

Kind	Minimum	Maximum	Note
Message type	0	65535	Representable as 16-bit unsigned integer (2 octets). The range of message data type ID usable with anonymous message transfers is further limited; more info in the chapter 4.
Service type	0	255	Representable as 8-bit unsigned integer (1 octet).

All UAVCAN nodes must use the same data type ID mapping for the standard data types, as defined by the default data type ID values provided for each of the standard data types. Since the standard data type ID mapping is immutable, all standard-compliant nodes can always use standard data types conflict-free.

Vendor-specific data types, however, do not enjoy the lack of conflict guarantee, because by virtue of being vendor-specific, such data types cannot use a global fixed agreed upon mapping like the standard data types do. As such, whenever vendor-specific data types are used, there is always a risk that different nodes may map different data types to the same data type ID.

It is the responsibility of the system integrator to ensure that if vendor-specific data types are used, the data type ID mappings are configured on all nodes identically. Vendors of UAVCAN equipment must provide the integrator with a way to change the data type ID of any vendor-specific data type leveraged by the node[19].

3.8 Standard and vendor-specific data types

3.8.1 Standard data type repository

The DSDL definitions of the standard data types are available in the official DSDL repository, which is linked from the project homepage at uavcan.org.

Information concerning development and maintenance of the standard DSDL definitions is available in the chapter 1.

3.8.2 Vendor-specific data types

Vendors must define their specific data types in a separate namespace, which should typically be named to match their company name. Separation of the vendor's definitions into a dedicated namespace ensures that no name conflicts will occur in systems that utilize vendor-specific data types from different providers. Note that, according to the naming requirements, the name of a DSDL namespace must start with an alphabetic character; therefore, a company whose name starts with a digit will have to resort to a mangled name, e.g. by moving the digits towards the end of the name, or by spelling the digits in English (e.g. 42 - `fortytwo`).

Defining vendor-specific data types within the standard namespace (`uavcan.`) is explicitly prohibited. The standard namespace will always be used only for standard data types.

Generally speaking, it is desirable for "generic" data types to be included into the standard set. Vendors should strive to design their data types as generic and as independent of their specific use cases as possible. The SI system of measurement units should be preferred; data type definitions that make unnecessary deviations from SI will not be accepted into the standard set.

[19]Nodes that fail to provide a way of altering the data type ID mapping for vendor-specific data types cannot be considered standard-compliant.

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.

4.1 The concept of transfer

A *transfer* is an act of data transmission between nodes. A transfer that is addressed to all nodes except the source node is a *broadcast transfer*. A transfer that is addressed to one particular node is a *unicast transfer*. UAVCAN defines the following types of transfers:

Message transfer - a broadcast transfer that contains a serialized message.

Service transfer - a unicast transfer that contains either a service request or a service response.

Both message and service transfers can be further distinguished between:

Single-frame transfer - 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.

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

The following properties are common to all types of transfers:

Table 4.1: Common transfer properties

Property	Description
Payload	The serialized data structure.
Data type ID	A numerical identifier that indicates how the data structure should be interpreted.
Data type major version number	Semantic major version number of the data type definition.
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.

4.1.1 Message broadcasting

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

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

In order to broadcast a message, the broadcasting node must have a node ID that is unique within the network. An exception applies to *anonymous message broadcasts*.

4.1.1.1 *Anonymous message broadcasting*

An anonymous message transfer is a transfer that can be sent from a node that does not have a node ID. This sort of message transfer is especially useful for *dynamic node ID allocation* (a high-level concept that is reviewed in detail in the 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 data 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, which in the case of anonymous message transfers is always assumed to be zero.

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.

The set of message type ID values that can be used with anonymous messages may be limited depending on the transport in use (section 4)[20]. It is guaranteed, however, that the message data type ID in the range from 0 to 7, inclusively, are always available for use with anonymous messages regardless of the transport in use.

Note that anonymous messages require specific arbitration rules and have restrictions on the acceptable data type ID values. The details are explained later in this chapter.

4.1.1.2 *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 data may opt for higher timeout values to account for the network congestion.

4.1.2 Service invocation

A service invocation sequence consists of two related service transfers:

Service request transfer - from the node that invokes the service - the *client* - to the node that provides the service - the *server*.

Service response transfer - once the *server node* receives the service request and processes it, it sends a response transfer back to the *client node*.

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.

[20]This 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.
Data type ID	See the table 4.1.
Data type major version number	See the table 4.1.
Source node ID	The node ID of the client (the invoking node).
Destination node ID	The node ID of the server (the invoked node).
Priority	See the table 4.1.
Transfer ID	An integer value that: 1. allows the server to distinguish the request from other requests from the same client; 2. allows the client to match the response with its request.

Table 4.3: Service response transfer properties

Property	Description
Payload	The serialized service response data structure.
Data type ID	Same value as in the request transfer.
Data type major version number	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.

4.1.2.1 *Service timing requirements*

Applications should 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.

4.1.3 Transfer prioritization

UAVCAN transfers are prioritized by means of the transfer priority property, which allows at least eight different priority levels for all types of transfers. The priority level mnemonics are specified in the following list. The mapping between the mnemonics and actual numeric identifiers is transport-dependent.

- Foreground (highest)
- Super
- Urgent
- High
- Normal

- Low
- Diagnostic
- Background (lowest)

Transfers with higher priority levels preempt transfers with lower priority levels, delaying their transmission until there are no more higher priority transfers to exchange.

4.2 Transfer emission

4.2.1 Transfer ID computation

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

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

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

Transport frame deduplication - for single-frame transfers, the transfer ID allows receiving nodes to work around the transport frame duplication problem[21] (multi-frame transfers combat the frame duplication problem using the toggle bit).

Multi-frame transfer reassembly - more info is provided in the 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.

The logic to compute the transfer ID relies 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 node, share the same data type ID, same data type major version number, and the same type. The properties that constitute a transfer descriptor are listed below:

- Transfer type (message broadcast, service request, etc.).
- Data type ID.
- Data type major version number.
- Source node ID.
- Destination node ID (only for unicast[22] transfers).

Every non-passive node must maintain a mapping from transfer descriptors to transfer ID counters. This mapping is referred to as the *transfer ID map*.

[21] This is a well-known issue that can be observed with certain transports such as CAN bus – a frame that appears valid to the receiver may under certain (rare) conditions appear invalid to the transmitter, triggering the latter to retransmit the frame, in which case it will be duplicated on the side of the receiver. Sequence counting mechanisms such as the transfer ID or the toggle bit (both of which are used in UAVCAN) allow applications to circumvent this problem.

[22] I.e., service requests and service responses.

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 publish certain transfers aperiodically or on an ad-hoc basis, thereby creating unused entries in the transfer ID map. In order to avoid keeping unused entries in the map, the nodes are allowed, but not required, to remove entries from the map that were not used for more than 2 seconds. Therefore, it is possible that a node may publish a transfer with an out-of-order transfer ID value, if the previous transfer of the same descriptor has been published more than 2 seconds earlier.

4.2.2 Single frame transfers

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

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

4.2.3 Multi-frame transfers

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

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

- Transfer CRC[23].
- 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 CRC value at the end of the transfer payload, and then emits the resulting byte set in chunks as an ordered sequence of transport frames (i.e. the last transport frame contains the last bytes of the payload and the transfer CRC). The data field of all transport frames of a multi-frame transfer, except the last one, must be fully utilized.

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.

4.2.3.1 Transfer CRC

The objective of the transfer CRC is to allow receiving nodes to validate correctness of multi-frame transfer reassembly. It should be understood that the transfer CRC is not intended for bit-level data integrity checks, as that must be managed by the transport layer implementation on a per-frame basis. 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 implemen-

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

tations, such as CAN FD, may require a short sequence of padding bytes to be added at the end of the transfer payload due to low granularity of the frame payload length property; in that case, the padding bytes are not to be included in the CRC computation.

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 this 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	`0xFFFF`
Polynomial	`0x1021`
Reverse	No
Output XOR	0
Residue	0
Check	$(49, 50, \ldots, 56, 57) \rightarrow$ `0x29B1`

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

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

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

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

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

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

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

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

4.2.3.2 *Toggle bit*

The toggle bit is a property defined at the transport frame level. Its purpose is to detect and avoid transport frame duplication errors in multi-frame transfers[24].

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

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

version detection.

4.2.4 Redundant interface support

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

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

4.3 Transfer reception

4.3.1 Transfer ID comparison

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

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

Consider the examples provided in the table 4.5.

The *half range* of transfer ID is 16.

Table 4.5: Transfer ID forward distance examples

A	B	F
0	0	0
0	5	5
5	0	27
31	30	31
31	0	1

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

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

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

constexpr std::uint8_t TransferIDBitLength = 5;

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

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

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

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

4.3.2 State variables

Nodes that receive transfers must keep a certain set of state variables for each transfer descriptor[25]. The set of state variables 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, however, that real implementations may resort to different architectures as long as the resulting behavior of the node observable at the protocol level is functionally equivalent.

The list of receiver state variables is provided in the table 4.6. Operations defined on receiver states are listed in the table 4.7; and the set of conditions defined for receiver states is provided in the table 4.8.

Whenever a node receives a transfer, it will query its receiver map for the matching transfer descriptor. If the matching state does not exist, the node will add a new uninitialized receiver state to the map. The node then will proceed with the procedure of *receiver state update*, which is defined below in this section.

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, once the state reaches the *transfer*

[25]The concept of *transfer descriptor* is explained in the section 4.2.1.

ID timeout condition.

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 background maintenance processes.

Table 4.6: Transfer reception state variables

State	Description
Transfer payload	Useful payload byte sequence; extended upon reception of transport frames.
Transfer ID	The transfer ID value of the current transfer or the next expected transfer. 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 leap.
Interface index	Only in the case of redundant transport interfaces.

Table 4.7: Transfer reception state operations

Operation	Description
Extension	Add newly received useful payload data to the current transfer payload state.
Restart	Reset the state variables to match the parameters of a new transfer. A reset can only be performed synchronously with the reception of a matching transport frame which is the first frame of a new transfer (e.g. the start of transfer flag is set). A reset operation includes at least the following: • Clearing (emptying) the transfer payload state. • Updating the transfer ID state with the actual transfer ID value from the new transfer. • Setting the toggle bit to its initial state (section 4.2.3.2). • Initializing the transfer timestamp with the reception timestamp from the transport frame. • Initializing the interface index (for nodes with redundant interfaces only).

Table 4.8: Transfer reception state conditions

Condition	Description
Uninitialized	The default condition, which indicates that the receiver state has not yet seen any transfers.
Transfer ID timeout	Last matching transfer was seen more than 2 seconds ago.
Interface switch allowed	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 has been successfully received more than T_{switch} seconds ago. The value of T_{switch} must not exceed 2 seconds. The actual value of T_{switch} can be either a constant chosen by the designer according to the application requirements (e.g., maximum recovery time in the event of an interface failure), or the protocol stack can estimate this value automatically by analyzing the transfer intervals.

4.3.3 State update in a redundant interface configuration

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

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

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

function receiveFrame(frame)
{
    // Resolving the state flags:
    tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
    same_iface := frame.iface_index == iface_index;
    first_frame := frame.start_of_transfer;
    non_wrapped_tid := computeForwardDistance(current_transfer_id, frame.transfer_id) < tid_half_range;
    not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;
    iface_switch_allowed := (frame.timestamp - this_transfer_timestamp) > iface_switch_delay;

    // Using the state flags from above, deciding whether we need to reset:
    need_restart :=
        (!initialized) or
        (tid_timed_out) or
        (same_iface and first_frame and not_previous_tid) or
        (iface_switch_allowed and first_frame and non_wrapped_tid);

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

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

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

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

    if (first_frame)
    {
        this_transfer_timestamp := frame.timestamp;
    }
    toggle := !toggle;
    payload.append(frame.data);

    if (frame.last_frame)
    {
        // CRC validation for multi-frame transfers is intentionally omitted for brevity
        processTransfer(payload, ...);

        current_transfer_id.increment();
        toggle := 0;
        payload.clear();
    }
}
```

4.3.4 State update in a non-redundant interface configuration

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

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

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

function receiveFrame(frame)
{
    // Resolving the state flags:
    tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
    first_frame := frame.start_of_transfer;
    not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;

    // Using the state flags from above, deciding whether we need to reset:
    need_restart :=
        (!initialized) or
        (tid_timed_out) or
        (first_frame and not_previous_tid);

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

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

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

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

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

    if (frame.last_frame)
    {
        // CRC validation for multi-frame transfers is intentionally omitted for brevity
        processTransfer(payload, ...);

        current_transfer_id.increment();
        toggle := 0;
        payload.clear();
    }
}
```

4.4 CAN bus transport layer specification

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

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

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

4.4.1 CAN ID structure

UAVCAN utilizes three different CAN ID formats for different types of transfers: message transfers, service transfers, and anonymous message transfers. The structure is summarized on the figure 4.1.

Bit	Service	Message	Anonymous message	Bit
28	Transfer priority	Transfer priority	Transfer priority	28
27				27
26				26
25	Service not message	Service not message	Reserved, required =0	25
24	Request not response	Message data type ID		24
23	Service data type ID			23
22				22
21			Message DTID modulo 8	21
20				20
19				19
18			Payload discriminator	18
17				17
16				16
15	Destination node ID			15
14				14
13				13
12				12
11				11
10				10
9				9
8	Source node ID	Source node ID	Source node ID	8
7				7
6				6
5				5
4				4
3				3
2				2
1	Data type major version number modulo 4	Data type major version number modulo 4	Data type major version number modulo 4	1
0				0
Bit	**Service**	**Message**	**Anonymous message**	**Bit**

Figure 4.1: CAN ID structure

The fields are described in detail in the following sections. The tables 4.9, 4.10, and 4.11 summarize the purpose of the fields and their permitted values for message transfers, anonymous message transfers, and service transfers, respectively. The following acronyms are used for brevity:

DTID - data type ID.

DTMVN - data type major version number.

Table 4.9: CAN ID fields for message transfers

Field	Width	Permitted values	Description
Transfer priority	3	[0, 7] (any)	Section 4.1.3.
Service not message	1	0	Always zero for message transfers.
Message DTID	16	[0, 65535] (any)	Data type ID of the encoded message data structure.
Source node ID	7	[1, 127]	Node ID of the origin.
Message DTMVN	2	[0, 3] (any)	Major version number of the data type, modulo 4.

Table 4.10: CAN ID fields for anonymous message transfers

Field	Width	Permitted values	Description
Transfer priority	3	[0, 7] (any)	Section 4.1.3.
Reserved field	4	0	Set to zero when emitting. When receiving, ignore the frame if this field is not zero.
Message DTID modulo 8	3	[0, 7] (any)	Three least significant bits of the data type ID of the encoded message data structure. Message types where DTID is greater than 7 cannot be used with anonymous message transfers.
Payload discriminator	10	[0, 1023] (any)	Used for CAN ID conflict avoidance; see section 4.4.1.5.
Source node ID	7	0	Set to zero. This field is used to distinguish anonymous message transfers from regular message transfers.
Message DTMVN	2	[0, 3] (any)	Major version number of the data type, modulo 4.

Table 4.11: CAN ID fields for service transfers

Field	Width	Permitted values	Description
Transfer priority	3	[0, 7] (any)	Section 4.1.3.
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 DTID	8	[0, 255] (any)	Data type ID of the encoded service data structure (request or response).
Destination node ID	7	[1, 127]	Node ID of the destination (i.e., server for requests, client for responses).
Source node ID	7	[1, 127]	Node ID of the origin (i.e., client for requests, server for responses).
Service DTMVN	2	[0, 3] (any)	Major version number of the data type, modulo 4.

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. Mnemonics for transfer priority levels are provided in the section 4.1.3.

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

Shall there be multiple transfers of different types at the same priority level contesting for the bus access, the following precedence is ensured, from higher priority to lower priority:

1. Message transfers.
2. Service response transfers.
3. Service request transfers.

Message transfers take precedence over service transfers because message broadcasting 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 data type ID: transfers with lower data type ID values preempt those with higher data type ID values.

4.4.1.2 *Data type ID*

A higher-level review of the concept of data type ID is available in the chapter 3.

For anonymous message transfers, the range of usable message type ID values is limited to [0, 7]; messages with data type ID outside of this range cannot be used with anonymous message transfers with this transport.

4.4.1.3 *Data type major version number*

As explained in the section 3.6, the difference between the lowest and the highest released major version numbers of any given data type may never exceed three.

Having made certain assumptions about the data type release cadence and the deprecation strategy, one can see that the transport layer can represent the data type major version number using a reduced modulo 4 representation; i.e., instead of carrying the whole version number, the transport layer can carry only the major version number modulo four:

$$V' = V \bmod 4 \Leftrightarrow V \geq 0$$

where V is the data type major version number and V' is its reduced representation used by the transport layer.

Taking advantage of the fact that the transport layer representation of the major data type version number belongs to [0, 3], UAVCAN uses only a two-bit wide field in the CAN identifier to represent the major version number.

The bit width saving measures described here warrant certain special handling rules for the major version number at the transport layer. When emitting a transfer, the node must compute the modulo of the major data type version number as described above, and populate the corresponding CAN ID field with the resulting value. When receiving a transfer, the node must search for an appropriate data type definition among the known definitions by looking for the one which has the same major version number modulus as the received modulus:

$$V_{\text{local}} \bmod 4 = V'$$

where V_{local} is the major version number of the locally available data type definition that will be used to process the transfer.

From the above description one can see that a data type mismatch will occur if the absolute difference between the major data type version used by the emitter and that used by the receiver exceeds three:

$$V_{\text{local}} \neq V \Leftrightarrow |V_{\text{local}} - V| \geq 4$$

However, due to the implicit deprecation policy defined in the section 3.6, the risk of version conflict is easy to avoid, and the limitation is therefore deemed acceptable.

4.4.1.4 *Node ID*

Valid values of node ID belong to the range [1, 127].

Node ID is represented by a 7-bit unsigned integer value; zero is reserved. A node ID of zero is used to represent either an unknown node or all nodes, depending on the context.

As such, for anonymous message transfers, the source node ID field is always set to zero. By observing a source node ID of zero, receiving nodes can distinguish anonymous message transfers from other types of transfers.

4.4.1.5 *Payload discriminator*

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 all share the same node ID of zero.

In order to work-around this problem, UAVCAN adds a *payload discriminator* to the CAN ID of anonymous message transfers, and defines special logic for handling CAN bus errors during transmission of anonymous frames.

The payload discriminator field must be filled with pseudorandom data whenever a node transmits an anonymous message transfer. The source of the pseudorandom data must be likely to produce different discriminator values for different data field values. A possible way of initializing the payload discriminator value is to apply the transfer CRC function (as defined in the section 4.2.3.1) to the contents of the anonymous message, and then use any 10 bits of the result. Nodes that adopt this approach will be using the same payload discriminator value for identical messages, which is acceptable since this will not trigger an error on the bus.

Since the discriminator is only 10 bits long, the probability of having multiple nodes emitting CAN frames with the same CAN ID but different data can exceed 0.1%, which is significant. Therefore, the protocol must account for possible errors on the CAN bus triggered by CAN ID collisions. In order to comply with this requirement, UAVCAN requires all nodes to immediately abort transmission of all anonymous transfers once an error on the CAN bus is detected. This measure allows the protocol to prevent the bus deadlock that may occur if the automatic retransmission on bus error is not suppressed.

4.4.2 **CAN frame data**

The CAN frame data field may contain the following data items, 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 4.12.

Table 4.12: Tail byte structure

Bit	Field	Single-frame transfers	Multi-frame transfers
7	Start of transfer	Always 1	First frame: 1, otherwise 0.
6	End of transfer	Always 1	Last frame: 1, otherwise 0.
5	Toggle bit	Always 1	First frame: 1, then alternates; section 4.2.3.2.
4	Transfer ID	Modulo 32 (range [0, 31]) section 4.2.1	
3			
2			
1			
0		(least significant bit)	

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

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

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

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

4.4.2.2 *Padding bytes*

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

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

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

Usage of padding bytes implies that when a serialized message is being deserialized by a receiving node, the byte sequence used for deserialization may be longer than the actual byte sequence generated by the emitting node during serialization. Therefore, nodes must ignore the trailing unused data bytes at the end of serialized byte sequences; a length mismatch is only to be considered an error if the received byte sequence is shorter than expected by the deserialization routine.

4.4.2.3 *Single-frame transfers*

For single-frame transfers, the data field of the CAN frame contains two or three segments: the useful payload (which is the serialized 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 4.13.

Table 4.13: 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.

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 previous 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. Therefore, UAVCAN implementations must be able to correctly process 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 4.14.

Table 4.14: 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.
L_{payload}	1	Tail byte.

The last CAN frame of a multi-frame transfer contains one or two additional segments: the padding bytes (if necessary) and the transfer CRC. The padding rules are identical to those of single-frame transfers. The transfer CRC is to be allocated in the big-endian byte order[26] immediately before the tail byte. The resulting data field segmentation is shown in the table 4.15.

Table 4.15: 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.

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

4.4.3 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 time-stamping*

Certain advanced features of UAVCAN may require the driver to time-stamp outgoing transport frames, e.g., the time synchronization feature. A sensible approach to transmission time-stamping is built around the concept of *loop-back frames*, which is described here.

If the application needs to time-stamp 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 time-stamp 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*

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:

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

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

    return chosen_mailbox
}
```

4.4.3.4 *Automatic hardware acceptance filter configuration*

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

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

Most CAN controller vendors implement mask-based filters, where the behavior of each filter is defined by two parameters: the mask M and the reference ID R. Then, such filter accepts only those CAN frames for which the following bitwise logical condition holds true[27]:

$$((X \wedge M) \oplus R) \leftrightarrow 0$$

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

[27] Notation: $\wedge$ – bitwise logical AND, $\oplus$ – bitwise logical XOR, $\neg$ – bitwise logical NOT.

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.

5 Application layer

5.1 Application-level conventions

5.2 Application-level functions

6 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.

6.1 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.

6.1.1 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 6.1 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, so that the device can be connected to the bus without the need to use T-connectors. T-connectors should be avoided when possible because generally they add an extra point of failure, increase the stub length, weight, and often require more complex and expensive wiring harnesses.

Table 6.1: 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

6.1.1.1 *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 figure 6.1.

Advantages	**Disadvantages**
• Highest level of compatibility with the existing commercial off the shelf (COTS) hardware. Connectors, cables, termination plugs, and other components can be easily purchased from many different vendors. • High-reliability options are available from multiple vendors. • Low-cost options are available from multiple vendors. • Both PCB mounted and panel mounted types are available.	D-Subminiature connectors are the largest connector type defined by UAVCAN. Due to its significant size and weight, it may be unsuitable for many vehicular applications.

The UAVCAN D-Sub connector is based on the industry-standard **D-Sub DE-9** (9-circuit) connector type. Devices are equipped with the male plug connector type mounted on the panel or on the PCB, and the cables are equipped with the female socket connectors on both ends (see the figure 6.1).

If the device uses two parallel connectors per CAN bus interface (as recommended), then all of the lines of the paired connectors, including those that are not used by the current specification, must be interconnected one to one. This will ensure compatibility with future revisions of the specification that make use of currently unused circuits of the connector.

The CAN physical layer standard that can be used with this connector type is ISO 11898-2[28].

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 6.2 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.

[28] Also known as *high-speed CAN*.

Table 6.2: 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.

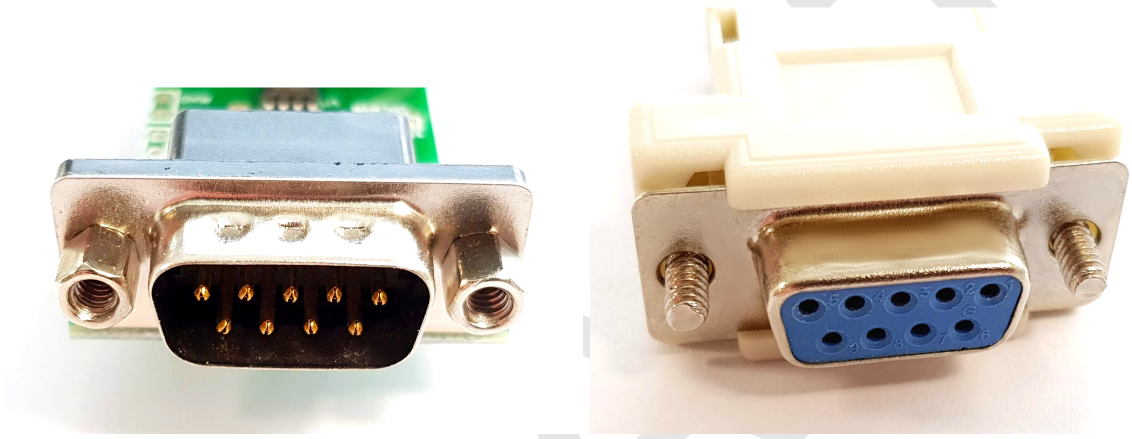


Figure 6.1: UAVCAN D-Sub connector pair example: device connector (left) and cable (right).

6.1.1.2 *UAVCAN M8 connector*

The UAVCAN M8 connector type is based on the generic circular M8 connector type, shown on the figure 6.2. This is a popular industry-standard connector, and there are many vendors that manufacture compatible components: connectors, cables, termination plugs, T-connectors, and so on. The pinning, physical layer, and supply voltages used in this connector type are compatible with CiA 103 (CANopen) and some other CAN bus standards.

The M8 connector is preferred for most UAVCAN applications (it should be the default choice, except when there are specific reasons to select another standard connector type).

Advantages	**Disadvantages**
• Compatibility with existing COTS hardware. Connectors, cables, termination plugs, and other components can be purchased from many different vendors. • High-reliability options are available from multiple vendors. • Low-cost options are available from multiple vendors. • Reasonably compact. M8 connectors are much smaller than D-Sub. • PCB mounted and panel mounted types are available.	• M8 connectors may be a poor fit for applications that have severe weight and space constraints. • The level of adoption in the industry is noticeably lower than that of the D-Sub connector type.

The UAVCAN M8 connector is based on the industry-standard **circular M8 B-coded 5-circuit** connector type. Devices are equipped with the male plug connector type mounted on the panel or on the PCB, and the cables are equipped with the female socket connectors on both ends (see the figure 6.2). *Do not confuse A-coded and B-coded M8 connectors – they are not mutually compatible*.

The CAN physical layer standard that can be used with this connector type is ISO 11898-2[29].

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.

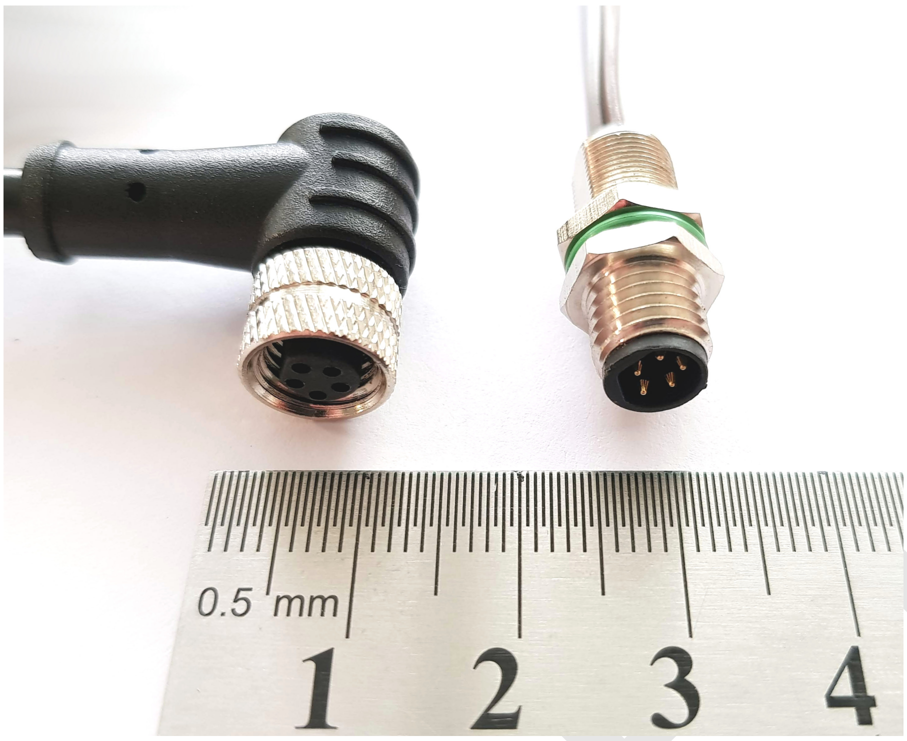
Devices that are powered from the bus should expect 18.0–30.0 V on the bus power line. The maximum recommended current draw from the bus is 0.5 A per device.

The table 6.3 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 6.3: 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	

[29] Also known as *high-speed CAN*.



Example connectors: female socket cable (left) and male plug device connector (right). Different connector types are available from various vendors: PCB mounted, panel mounted; straight cables, angled cables, etc.
Figure 6.2: UAVCAN M8 connector pair example.

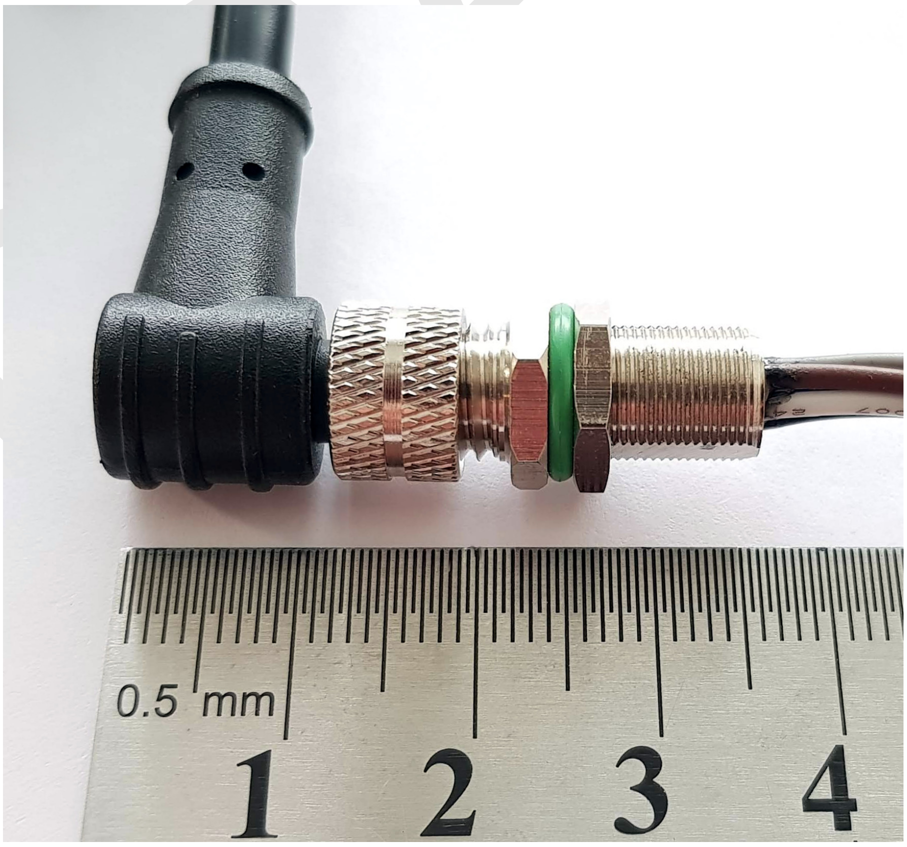


Figure 6.3: UAVCAN M8 assembled connector pair example.

6.1.1.3 *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 6.4.

The Micro connector is compatible with the Dronecode Autopilot Connector Standard. This connector type is recommended for small UAV and nanosatellites. It is also the recommended connector for attaching external panel-mounted connectors (such as the M8 or D-Sub types) to the PCB inside the enclosure.

Advantages	**Disadvantages**
• Extremely compact, low-profile. The PCB footprint is under 9âIJŢ5 millimeters. • Secure positive lock ensures that the connection will not self-disconnect when exposed to vibrations. • Low-cost, easy to stock.	• Board-level connections only. No panel-mounted options available. • No shielding available. • Not suitable for safety-critical hardware.

The UAVCAN Micro connector is based on the proprietary **JST GH 4-circuit** connector type.

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 EMI[30]; 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 6.4 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 6.4: 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	

[30] Electromagnetic interference.

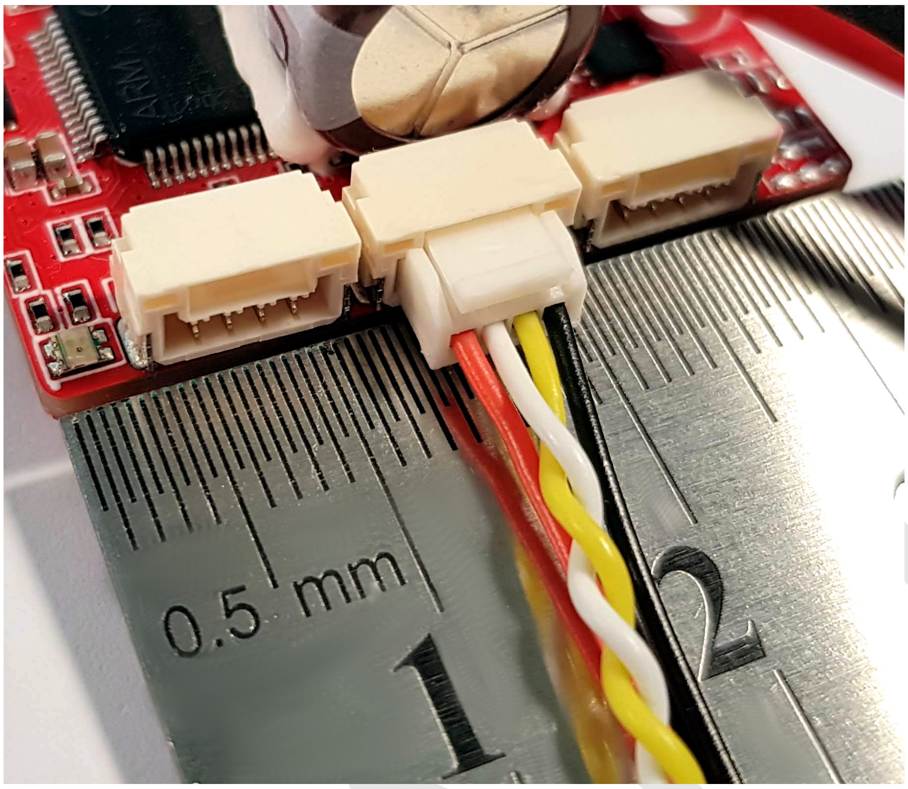


Figure 6.4: UAVCAN Micro right-angle connectors with a twisted pair patch cable connected.

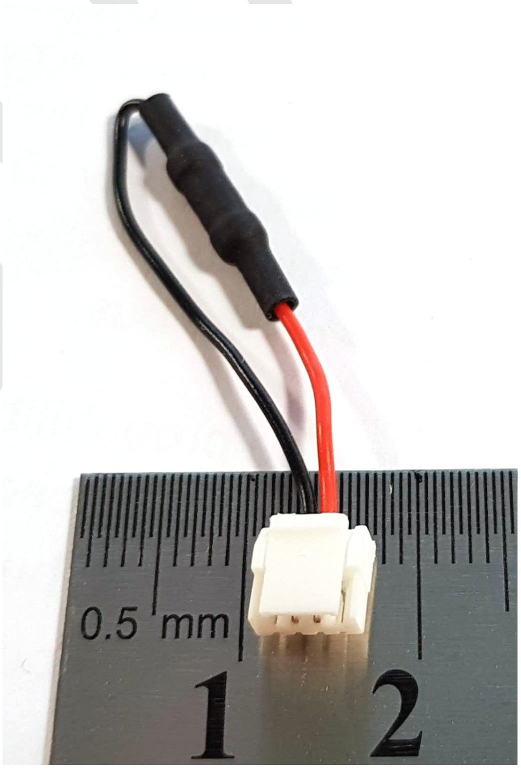


Figure 6.5: UAVCAN Micro CAN bus termination plug.

6.1.2 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.

6.1.2.1 CAN 2.0

This section is dedicated to the legacy CAN 2.0 protocol.

The table 6.5 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 6.5: 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.

6.1.2.2 CAN FD

This section will be populated in a later revision of the document.

6.2 Hardware design recommendations

This section contains certain generic hardware design recommendations that are agnostic of a particular physical layer implementation.

6.2.1 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 6.6 shows a doubly redundant bus transport as an example.

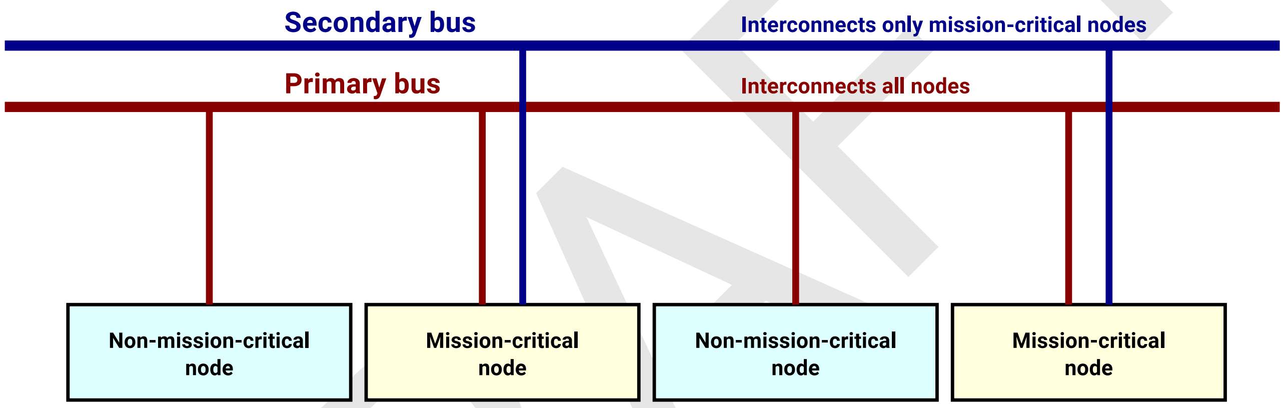

Secondary bus
Interconnects only mission-critical nodes
Primary bus
Interconnects all nodes
Non-mission-critical node
Mission-critical node
Non-mission-critical node
Mission-critical node

Figure 6.6: Non-uniform transport redundancy.

6.2.2 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.

6.2.2.1 Power sinking nodes

This section applies to nodes that draw power from the network.

Each power input must 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 must 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.

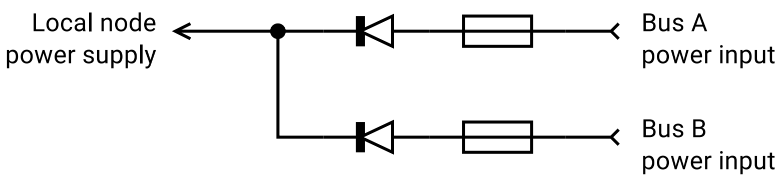


Figure 6.7: Simplified conceptual power sinking node design schematic.

6.2.2.2 *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.

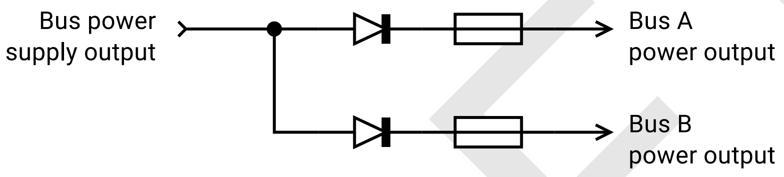


Figure 6.8: Simplified conceptual power sourcing node design schematic.