Specification v1.0

Revision 2018.07.23

Overview

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

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.
- Doubly- or triply- redundant CAN bus support.
- Supports high-precision network-wide time synchronization.
- The specification and high quality reference implementations in popular programming languages are free and open source.

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.

[1]Remote procedure call

Table of contents

List of figures

List of tables

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.

The reader should have a solid understanding of the main concepts and operating principles of the CAN bus.

1.1 Core design goals

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

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

Nodes can exchange long payloads - Nodes must be provided with a simple way to exchange large data structures that cannot fit into a single CAN[2] 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 - This is a common requirement for safety-critical applications.

High throughput, low latency communication - Applications that are dependent on high-frequency, hard real-time control loops, require a low-latency, high-throughput communication method.

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 should 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 Specification update and approval process

The UAVCAN development team is charged with advancing the specification 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 one of the cornerstone concepts of the specification (the data structure description language (DSDL) and related concepts are described in section 3). Within the same major version, the specification can be extended only in the following ways:

[2] Or CAN FD. Here and in the following parts of the specification, CAN also implies CAN FD, unless specifically noted otherwise.
[3] Please refer to uavcan.org.

- A new data type can be added, possibly with 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 the modification doesn't break backward compatibility.
- A new version of an existing data type can be added.
- 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 announced via the discussion forum, and indicated in the form of a comment in its DSDL definition.

A link to the repository containing the set of default DSDL definitions can be found on the official website[4].

1.3 Referenced sources

The UAVCAN specification contains references to the following sources:

- CiA 801 - Application note - Automatic bit rate detection.
- CiA 103 - Intrinsically safe capable physical layer.
- CiA 303 - Recommendation - Part 1: Cabling and connector pin assignment.
- IEEE 754 - Standard for binary floating-point arithmetic.
- ISO 11898-1 - Controller area network (CAN) - Part 1: Data link layer and physical signaling.
- ISO 11898-2 - Controller area network (CAN) - Part 2: High-speed medium access unit.
- ISO/IEC 10646 - Universal Coded Character Set (UCS).
- ISO/IEC 14882 - Programming Language C++.
- "Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus", M. Gergeleit and H. Streich.
- "In Search of an Understandable Consensus Algorithm (Extended Version)", Diego Ongaro and John Ousterhout.
- semver.org - Semantic versioning specification.

[4]uavcan.org

2 Basic concepts

UAVCAN is a lightweight protocol designed to provide a highly reliable communication method for aerospace and robotic applications via the CAN bus. A UAVCAN network is a decentralized peer network, where each peer (node) has a unique numeric identifier - *node ID*. 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[5].

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 CAN bus transport layer (chapter 4), which implements automatic decomposition/reassembly of long transfers into/from several CAN frames, allowing nodes to exchange data structures of arbitrary size.

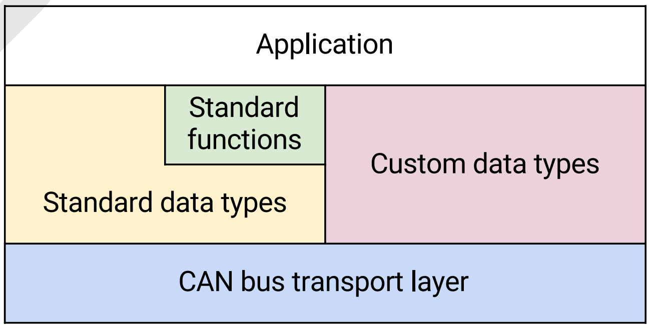


Figure 2.1: UAVCAN architectural diagram.

[5] Like remote procedure call (RPC).

2.1 Message broadcasting

Message broadcasting refers to the transmission of a serialized data structure over the CAN bus to other nodes. This is the primary data exchange mechanism used in UAVCAN. 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 type of message from a given node

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.

This kind of data exchange is useful during initial configuration of the node, particularly during the dynamic node ID allocation procedure[6].

Anonymous messages cannot be decomposed into multiple CAN frames, meaning that their payload capacity is limited to that of a single CAN 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:

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.

[6] This is an optional feature.

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 description
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 to this service from a given node

Both request and response contain exactly the same values for all 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 CAN bus. 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[7]:

```
[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`.

[7] 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. 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.

3.2.3 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	Binary format
`bool`	1	`bool` (can be optimized for bit arrays)	$\{0, 1\}$	One bit
`intX`	$2 \le X \le 64$	`int8_t`, `int16_t`, `int32_t`, `int64_t`	$[-\frac{2^X}{2}, \frac{2^X}{2} - 1]$	Two's complement
`uintX`	$2 \le X \le 64$	`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`	$[0, 2^X - 1]$	
`float16`	16	`float`	± 65504	IEEE 754 binary16
`float32`	32	`float`	Approx. $\pm 10^{39}$	IEEE 754 binary32
`float64`	64	`double`	Approx. $\pm 10^{308}$	IEEE 754 binary64
`voidX`	$1 \le X \le 64$	N/A	N/A	X zero/ignored bits

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[8] 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[9]. 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[10]; however, the rules and patterns of such padding fall out of the scope of the DSDL specification.

3.5.1.1 *Example*

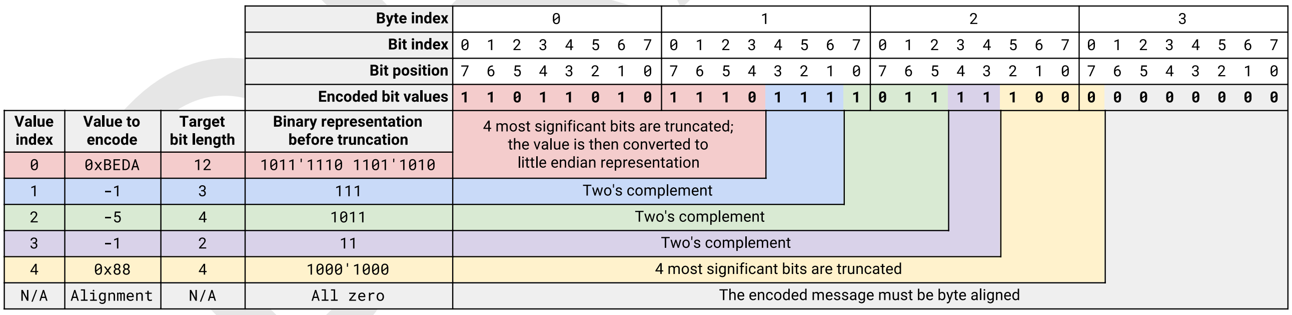
Consider the following data type definition:

```
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 encoded byte sequence is shown on the figure 3.1.



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.

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

Table 3.2: Scalar value serialization

Type	Bit length	Binary format
`bool`	1	Single bit
`intX`	X	Two's complement signed integer
`uintX`	X	Plain bits
`float16`	16	IEEE 754 binary16
`float32`	32	IEEE 754 binary32
`float64`	64	IEEE 754 binary64
`voidX`	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 encoded as a plain sequence of items, with each item encoded independently in place, with no alignment. No extra data is added.

Essentially, a fixed-size array of size X elements will be encoded 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 encoded as a sequence of encoded 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 encoded 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 encoded 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 encoded byte sequence would be (in binary):

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

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, an *encoded representation of A* is a sequence of data fields joined into a bit string according to the DSDL definition of the data structure A.

A structure definition A is bit-compatible with a structure definition B if any valid encoded representation of B is also a valid encoded representation of A. A and B are said to be *mutually compatible* if the sets of all possible valid encoded representations of A and B are identical.

3.6.2.2 Example

A *fixed-size data structure* is a structure that does not contain dynamic arrays or other structures that contain dynamic arrays within themselves. As such, the bit length of an encoded 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 encoded 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[11], and the array type is a fixed-size structure[12], 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 valid encoded representations of the second definition is a subset of those of the first definition. Possible encoded 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.

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

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.

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, and A is bit-compatible with B.

Because of the dependency on bit compatibility, the property of semantic compatibility is non-commutative.

3.6.3.2 Example

Despite using different binary layouts, the following two definitions are semantically 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[13].

3.6.4 Data type versioning

3.6.4.1 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.

[13]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.

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.

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

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 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 *Example*

Suppose a vendor *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  = 35768
# 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

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

float32 power_consumption_0     # Power consumption by the redundant PSU 0 [watt]
float32 power_consumption_1     # likewise for PSU 1
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  = 35768
# 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[14] 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

float16 internal_temperature    # [kelvin]
float16 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
```

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:

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

```
# 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 independent sets of data type identifiers: one for message data types and the other for service data types. Each 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.

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[15].

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.

[15]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 CAN bus transport layer

5 Application layer

5.1 Application-level conventions

5.2 Application-level functions

6 Hardware design recommendations