DCP SPECIFICATION

DISTRIBUTED CO-SIMULATION PROTOCOL

ACOSAR
ADVANCED CO-SIMULATION OPEN SYSTEM ARCHITECTURE

DOCUMENT TYPE:	**DELIVERABLE**
DELIVERABLE N^0:	**D6.1**
DISTRIBUTION LEVEL:	**PUBLIC**
DATE:	**14/09/2018**
VERSION:	**1.0.0-RELEASE CANDIDATE 2**

EDITOR: **MARTIN KRAMMER, KOMPETENZZENTRUM DAS VIRTUELLE FAHRZEUG FORSCHUNGSGESELLSCHAFT MBH**

AUTHORS: **CORE TEAM MEMBERS** (ALPHABETICAL ORDER)

KHALED ALEKEISH, ESI-ITI GMBH
NICOLAS AMRINGER, DSPACE GMBH
MARTIN BENEDIKT, KOMPETENZZENTRUM DAS VIRTUELLE FAHRZEUG FORSCHUNGSGESELLSCHAFT MBH
TORSTEN BLOCHWITZ, ESI-ITI GMBH
ISIDRO CORRAL, ROBERT BOSCH GMBH
MICHA DAMM-NORWIG, KS.MICRONOVA GMBH
CHRISTIAN KATER, LEIBNIZ UNIVERSITÄT HANNOVER
SERGE KLEIN, RWTH AACHEN UNIVERSITY
MARTIN KRAMMER, KOMPETENZZENTRUM DAS VIRTUELLE FAHRZEUG FORSCHUNGSGESELLSCHAFT MBH
STEFAN MATERNE, TWT GMBH
NATARAJAN NAGARAJAN, ETAS GMBH
ROBERTO RUVALCABA, TWT GMBH
VIKTOR SCHREIBER, UNIVERSITY OF ILMENAU
KLAUS SCHUCH, AVL LIST GMBH
TOMMY SPARBER, SPATH MICRO ELECTRONIC DESIGN GMBH
ANDREAS THUY, ETAS GMBH

PROJECT ACRONYM:	**ACOSAR**
PROJECT TITLE:	**ADVANCED CO-SIMULATION OPEN SYSTEM ARCHITECTURE**
ITEA PROJECT N^0:	**14004**
CHALLENGE:	**ENGINEERING**
PROJECT DURATION:	**01/09/2015 - 31/08/2018**
PROJECT WEBSITE:	**WWW.ACOSAR.EU**
COORDINATION:	**VIRTUAL VEHICLE RESEARCH CENTER (AT)**
PROJECT LEADER:	**DR. MARTIN BENEDIKT**

ITEA3

Executive summary

This document defines the Distributed Co-Simulation Protocol (DCP), version 1.0. The DCP is a platform and communication medium independent standard for the integration of real-time systems into simulation environments. The DCP development was initiated by VIRTUAL VEHICLE Research Center with the goal to make simulation based work flows more efficient and improve the integration of real-time systems. This initiative is supported by 16 OEMs, suppliers, tool providers, universities and research organizations from three different European countries.

Standardisation

This specification will be developed further by the newly founded Modelica Association Project (MAP) DCP. For further information see www.modelica.org. If you are interested to contribute to this specification please contact dcp@v2c2.at.

License of this document

Contents

1 Overview

Virtual system development is getting more and more important in a plenitude of industrial domains to reduce development times, stranded costs and time-to-market. Co-simulation is a particularly promising approach for interoperable and modular development. The Functional Mock-up Interface (FMI) defines a standardized specification for the integration of simulation models, tools and solvers. However, the coupling and integration of real-time systems into simulation environments (especially of distributed hardware-in-the-loop systems and simulations) still requires enormous effort.

The Distributed Co-Simulation Protocol (DCP) was developed in scope of the ACOSAR (Advanced Co-Simulation Open Systems Architecture) project. The DCP specifies a data model, a finite state machine, a set of protocol data units and a communication protocol. It is intended for the integration of real-time and/or non-real-time systems. It features a master-slave principle. Furthermore, it is defined independently of the underlying transport protocol and distinguishes between native and non-native DCP specifications. With this approach, support for additional transport protocols may be added in the future. The DCP specification also includes a default integration methodology.

The DCP specification document at hand can lead to a modular, considerably more flexible, as well as shorter system development process. The DCP is suitable for application in numerous industrial domains. Furthermore, it has the potential to enable new business models.

2 Properties and Guiding Ideas

In this section, properties are listed, and some principles are defined that guided the design of the DCP. Six central aspects drive the development: interoperability, integration, compatibility, communication, performance, and economy.

- Interoperability: The DCP defines a communication protocol intended for the exchange of simulation related information and data. It enables the interoperability of systems from different providers. This principle homogenizes the situation exposed when having a heterogeneous landscape of tools, protocols, and interfaces commonly found in today’s work environments. It further facilitates the deployment of co-simulation approaches across different computers, sites, and companies.
- Integration: The DCP enables the integration of distributed real-time systems and/or non-real-time systems into one common co-simulation scenario.
  The DCP operation follows a master-slave architecture. This type of architecture is beneficial because it ensures the integration of multiple DCP slaves into a common co-simulation scenario. This principle supports co-simulation of mixed systems, e.g. hardware setups and digital models. The DCP master-slave approach facilitates the addition and/or removal of single components without the need to stop any DCP slave. Consequently, one can switch between digital models and real hardware setups, and therefore accelerate the development process.
  Furthermore, a default integration methodology is provided. It demonstrates the interplay of different parts of the DCP specification. Furthermore, it also demonstrates the role of components that are not part of this DCP specification.
- Compatibility: The DCP is defined in a way such that it supports the integration of FMI based systems within DCP slaves. This applies to FMI for Model Exchange as well as FMI for Co-Simulation. The DCP state machine is designed that it matches operations defined in the state machine of the FMI. Furthermore, the DCP slave description file is aligned to the model description file of the FMI. The data types defined in the DCP slave description file may also be converted to FMI compatible data types. This principle supports FMI-based simulation models, considering the fact that FMI is one of the most common co-simulation standards today.
  Whereas the FMI represents an application programming interface (API), the DCP represents a communication protocol. Therefore, it becomes possible to integrate various kinds of systems. The DCP specification is suitable for a broad range of computing platforms. It may be implemented on hardware as well as in software. Typical examples are middleware, runtime environments, (virtualized) operating systems, electronic control units, FPGAs, and many more.
- Communication: The DCP enables simulation data exchange by a variety of communication systems and transport protocols. The DCP specification refrains from further specification of the communication medium. This attribute underscores the underlying design principle that the choice of the communication medium must be as convenient as possible for the end-user. DCP abstracts from the most common communication systems. As of today, supported communication systems and transport protocols include UDP/IPv4, Bluetooth, USB, and CAN.
  Modern system development processes require the exchange of many different data types. The DCP specification supports the transmission of data type primitives, vectors, binary data, and strings. Finally, the DCP specification offers a dedicated safe-state. It may be used to provide the possibility to implement mechanisms for protection of operators and the involved hardware. The safety mechanisms themselves are not inherent to the DCP specification.
- Performance: Distributed real-time and non-real-time co-simulation requires high performance of data exchange. For that reason, the exchanged simulation data at runtime does not contain any overhead data, like signal names or value references. The DCP supports data exchange between slaves via the co-simulation master, as well as direct slave to slave data exchange. The DCP master is free to define either a number of short data segments, or all data at once for the exchange of simulation data. This also depends on the capabilities of the communication medium.

- Economy: The DCP specification is intended to contribute to the following economic landmarks. First, it helps to reduce development time. This is achieved by independent design, development, and test of each individual subsystem. Therefore, the development process can be parallelized. Only the final integration happens collaboratively. Negotiations between system suppliers and integrators can be kept to a minimum. Second, the DCP specification is independent of any computing platform, which can decrease computing costs., Third, a shortened time-to-market can be achieved by efficiently setting up and running an increased number of test cases. Due to the open manner and free availability of the DCP specification document, a vivid and active DCP community distributes the DCP specification document into different application domains. The creation of business opportunities, especially for smaller companies, also emerges from this community. Finally, mutual support and exchange of experience will drive future development of the DCP.

3 Protocol Specification

3.1 Basic Definitions

The DCP is a platform-independent protocol which enables communication and data exchange for co-simulation, between a multitude of different computing platforms, operating systems and software. This section defines the data types supported by the DCP and their encodings to enable interoperability between these systems.

3.1.1 Keywords

Unless noted otherwise, the meaning of keywords (must, must not, should, …) as stated in Appendix A of this document applies.

3.1.2 Version Descriptor

This DCP specification utilizes the following version descriptor numbering scheme. See also section 5.3.

- `dcpMajorVersion`: First level version number. Indicates a major specification release that is relevant to compliant implementation.
- `dcpMinorVersion`: Second level version number. Indicates a minor specification release that is relevant to compliant implementation.
- `dcpMaintenanceVersion`: Third level version number. Indicates a specification release that is not relevant to compliant implementation.

3.1.3 DCP Slave

A DCP slave is either a simulation model or a real-time system on a ready-to-run execution platform that is accessible, via DCP, over a given supported communication medium. All static information related to a DCP slave is stored in a text file in XML format. The latter file is called DCP slave description. See chapter 5 for details.

3.1.4 Master-Slave Architecture

Exactly one DCP master may control at least one DCP slave. The DCP master is the only one to send DCP request PDUs within a single scenario. After registration, one DCP slave shall communicate with exactly one DCP master.

This DCP specification is intended for the realization of a DCP slave. It does not explicitly specify how a DCP master must be designed. A DCP master provider needs to ensure that its DCP master is able to correctly operate with at least one DCP slave according to this DCP specification.

3.1.5 State Machine

Each DCP slave internally implements a state machine, where a transition refers to a change of a state. Transitions can be triggered by PDUs. Details are given in section 3.2. At any given instant of time a DCP slave is in exactly one state. This assumes that transitions are instantaneous.

3.1.6 Protocol Data Units

DCP slaves communicate by using Protocol Data Units, short PDU. In general, a DCP slave must be capable of sending and receiving such PDUs. Available PDUs within DCP are organized in PDU families which are named Request, Response, Notification and Data. The Request PDUs consist of configuration request (CFG), state change request (STC) and information request (INF) PDUs. The Response (RSP) PDUs together with Request PDUs represent the family of Control PDUs. See also section 3.3.

3.1.7 Number Representation

All numbers given in this DCP specification document must be interpreted as decimal, if no prefix is used. Hexadecimal values are always indicated with the prefix 0x. If a binary number appears outside a table, binary numbers are indicated with the prefix 0b.

3.1.8 Indices

All indices and positions start at 0 (“zero”) unless stated otherwise.

3.1.9 Data Types

The supported data types of the DCP are defined in Table 1. Each data type is assigned a unique identifier (ID).

Data type	ID$_{hex}$
uint8	0x0
uint16	0x1
uint32	0x2
uint64	0x3
int8	0x4
int16	0x5
int32	0x6
int64	0x7
float32	0x8
float64	0x9
string	0xA
binary	0xB

Table 1: Supported data types of the DCP

3.1.10 Byte Order

The byte order considered for this entire DCP specification document is little endian, unless explicitly noted otherwise.

3.1.11 Data Type Encoding

3.1.11.1 Integer Numbers

- Unsigned integers (data types `uint8, uint16, uint32 and uint64`) are transferred as unsigned binary numbers in little endian byte order. The number of bits used to store the integer is defined by its suffix, e. g. 8 bits for `uint8`.
- Signed integers (data types `int8, int16, int32` and `int64`) are transferred as binary numbers in two's complement representation in little endian byte order. The required number of bits in memory for storing the integer is defined by the suffix, e. g. 8 bits for `int8`.
- Table 2 illustrates both the binary and the representation of the sample number i=-89498498 as `int32` in PDUs.

Binary	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0
Hex	0xFA								0xAA								0x5C								0x7E							
	MSB																															LSB

Position	n								n + 1								n + 2								n + 3							
DAT_input_output$_{Bin}$	0	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	1	0	1	1	1	1	1	0	1	0
DAT_input_output$_{Hex}$	0x7E								0x5C								0xAA								0xFA							

🟦 Sign 🟩 Binary Values

Table 2: int32 representation

3.1.11.2 Floating Point Numbers

32 bit floating point numbers (data type `float32`) are transferred in binary32 format, as defined in [1], in little endian byte order:

- The binary value is built from MSB to LSB by the following: Sign (1 bit), Exponent (8 bit), and Mantissa (23 bit).

64 bit double values (data type `float64`) are transferred in binary64 format, as defined in [1], in little endian byte order:

- The binary value is built from MSB to LSB by the following: Sign (1 bit), Exponent (11 bit), and Fraction (53 bit). This binary value is transferred in little endian byte order.
- Table 3 illustrates both the binary and the representation of the sample number f=7256.2568359375 as `float32` in PDUs.

Binary	0	1	0	0	0	1	0	1	1	1	1	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0	0	0	1	1	1	0
Hex	45								E2								C2								0E							
	MSB																															LSB

Position	n								n + 1								n + 2								n + 3							
DAT_input_output$_{Bin}$	0	0	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	0	1
DAT_input_output$_{Hex}$	0E								C2								E2								45							

🟦 Sign 🟩 Exponent 🟥 Fraction

Table 3: `float32` representation

See Appendix for other examples.

3.1.11.3 Binary

The DCP offers a binary data type (`binary`) to transmit arbitrary information. The binary representation consists of an unsigned integer (`uint16`) that specifies the length in bytes of the actual data, followed by the binary data itself. The data is transmitted as given without changing the order of its bits. Thus, the maximum length of a data is set to 65535 bytes.

Note: This general DCP specification does not define PDU fragmentation or splitting.

The example given in Table 4 and Table 5 shows the encoding of a four byte data sequence in `binary` data type. The actual data is given in Table 4, whereas in Table 5 the PDU representation of the payload is shown. The total length of the payload is 6 bytes, the first two bytes store an integer value (`uint16`) indicating the length (4 bytes) of the actual data.

Data Binary	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1	0	0	1	0
Data Hex	39								E6								29								D2							
Byte index	0								1								2								3							

Table 4: `binary` data type example

The payload is then encoded as shown in Table 5.

Position	n								n+1							
PDU$_{Bin}$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

PDU_{Hex}	0x04	0x00

Position	n+2	n+3	n+4	n+5
PDU_{Bin}	0 0 1 1 1 0 0 1	1 1 1 0 0 1 1 0	0 0 1 0 1 0 0 1	1 1 0 1 0 0 1 0
PDU_{Hex}	0x39	0xE6	0x29	0xD2

Table 5: Binary data type representation.

Note: A maximum length in bytes may be specified in DCP slave description by setting the maxSize attribute.

3.1.11.4 Strings

In general, the string data type is encoded in the same way as the binary data type. Strings are of variable length and are not terminated in any way. However, the specified character encoding for strings is UTF-8 [2].

Note: UTF-8 strings are handled byte-wise.

Note: A maximum length in bytes may be specified in DCP slave description by setting the maxSize attribute. Also note that the length in bytes does not necessarily match the number of encoded characters in the string.

Note: These definitions apply to protocol data units (PDUs, as defined in section 3.3) only.

The following Table 6 illustrates the encoding of the word "beef" (0x62, 0x65, 0x65, and 0x66). The payload is then encoded as shown in Table 7.

Data Binary	0 1 1 0 0 0 1 0	0 1 1 0 0 1 0 1	0 1 1 0 0 1 0 1	0 1 1 0 0 1 1 0
Data Hex, UTF-8	0x62	0x65	0x65	0x66
Byte index	0	1	2	3

Table 6: String data type example

Position	n	n+1
$DAT\_input\_output_{Bin}$	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0
$DAT\_input\_output_{Hex}$	0x04	0x00

Position	n+2	n+3	n+4	n+5
$DAT\_input\_output_{Bin, UTF-8}$	0 1 1 0 0 0 1 0	0 1 1 0 0 1 0 1	0 1 1 0 0 1 0 1	0 1 1 0 0 1 1 0
$DAT\_input\_output_{Hex, UTF-8}$	0x62	0x65	0x65	0x66

Table 7: String data type representation

3.1.12 Timing

DCP does not include mechanisms for time synchronization. If such mechanisms are needed, existing mechanism for time synchronization between nodes shall be used. A typical example for such a mechanism can be found in [3].

3.1.13 Notion of Time

3.1.13.1 Absolute Time

The absolute time is the newtonian time represented by a UNIX time stamp in UTC format. It is defined in seconds since January 1st, 1970, 00:00:00 UTC, minus the number of leap seconds from that date till now.

Note: This is also referred as epoch.

3.1.13.2 Simulation Time

The simulation time is the time value to which simulation models inside DCP slaves refer to.

3.1.14 Operating Modes

The DCP defines three different operating modes targeting the real-time properties specified in the following sections. A DCP slave must support at least one of them. Table 8 specifies the operating modes enumeration.

Operating mode	op_mode$_{hex}$
HRT	0x00
SRT	0x01
NRT	0x02

Table 8: Operating modes enumeration

The DCP slave is informed by the master about the chosen operating mode (one of HRT, SRT, NRT).

> *Note: For native DCP (see section 3.1.19), this is achieved via STC_register PDU (see section 3.3.6).*

3.1.14.1 Hard Real-Time (HRT)

All deadlines for all out ports must be met. Simulation time is synchronous to absolute time. In case of any deviations, the DCP slave transitions to the error state.

> *Note: Synchronous means that one unit of elapsed absolute time corresponds to the same unit of simulation time.*

3.1.14.2 Soft Real-Time (SRT)

It depends on the application if and how SRT DCP slaves are integrated into scenarios. The DCP slave tries to meet deadlines for all out ports. If deadlines are not met, the DCP slave continues operation. Simulation time should be synchronous to absolute time. It depends on the application, if and when the DCP slave signals an error.

3.1.14.3 Non-Real-Time (NRT)

Simulation time is independent from absolute time. It can be faster or slower. Reception of PDU STC_do_step (see section 3.3.7.6) is required.

3.1.15 Time Resolution

One atomic time step, i.e. the resolution, is defined as a fraction of two integer values `numerator` and `denominator`. It is set by the DCP master. For native DCP it is rolled out via PDU `CFG_time_res` in state `CONFIGURATION` (see section 3.2). The unit of the fraction is seconds. Possible values for the communication are defined in the DCP slave description, where either a valid range is specified or a list of valid values is provided.

3.1.16 Communication Step Size

The communication step size is defined as follows:

$$stepsize_{communication} = \frac{numerator}{denominator} \cdot steps$$

where `numerator` divided by `denominator` represents the resolution and `steps` represents the integer number of resolution intervals. The minimum value for `steps` is 1.
If the communication step size for an output should be fixed, then both the attributes resolution and `steps` need to be set to `fixed` in the DCP slave description.

For operating modes HRT and SRT, `steps` is configured via PDU `CFG_steps` (see section 3.3.7.14) by the DCP master in state `CONFIGURATION`.
For the operating mode NRT, `steps` is given in each PDU `STC_do_step` (see section 3.3.7.6).

3.1.17 Variables

All variable values (inputs, outputs, parameters, structural parameters) of a DCP slave are identified with a variable handle called *value reference* (abbreviated `vr`). This handle is defined in the DCP slave description file as attribute `valueReference` in element `Variable`. See section 5.12.2 for details.

3.1.17.1 Variable Naming Convention

Within the DCP slave description the attribute `variableNamingConvention` of element `dcpDescription` defines the convention how the variable names are constructed. This information may then be used by the simulation environment for structuring.
Possible options are given in Table 9.

> *Note: This is based on FMI 2.0.*

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>flat</td><td>name = Unicode-char { Unicode-char }
Unicode-char = any Unicode character without carriage return (0x0D), line feed (0x0A) nor tab (0x09)

This definition is identical to xs:normalizedString used in the specification of FMI.
The names shall be unique, non-empty strings are not allowed.</td></tr>
<tr><td>structured</td><td>Structured names are hierarchically organized and use "." as a separator between hierarchies. A name consists of "_", letters and digits or may consist of any characters enclosed in single apostrophes. A name may identify an array element on every hierarchical level using square brackets "[...]" to identify the respective array index.

In the following definitions, an extended Backus-Naur form (EBNF) [4] is used.

The precise syntax is:
name = identifier | "der(" identifier ["," unsignedInteger] ")"
identifier = B-name [arrayIndices] {"." B-name [arrayIndices] }
B-name = nondigit{digit|nondigit}|Q-name
nondigit = "_" | letters "a" to "z" | letters "A" to "Z"
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Q-name = "'" (Q-char | escape) { Q-char | escape } "'"
Q-char = nondigit | digit | "!" | "#" | "$" | "%" | "&" | "(" | ")" |
"*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | ">" | "=" | "?"
| "@" | "[" | "]" | "^" | "{" | "}" | "|" | "~" | " "
escape = "\'" | "\"" | "\?" | "\\" | "\a" | "\b" |
"\f" | "\n" | "\r" | "\t" | "\v"
arrayIndices = "[" unsignedInteger {"," unsignedInteger} "]"
unsignedInteger = digit { digit }

Note: This definition is identical to the syntax of an identifier in Modelica version 3.2.

The tree of names must be mapped to an ordered list of structured variable names in depth-first order.</td></tr>
</table>

	Example: *vehicle* *transmission* *ratio* *outputSpeed* *engine* *inputSpeed* *temperature* *is mapped to the following list of structured variable names:* *vehicle.transmission.ratio* *vehicle.transmission.outputSpeed* *vehicle.engine.inputSpeed* *vehicle.engine.temperature* *Note: No further restrictions apply (e.g., no alphabetical sort on same hierarchical level)* Variables representing array elements must be given in a consecutive sequence. Elements of multi-dimensional arrays are ordered according to row major order, that is elements of the last index are given in sequence. *For example, elements of the vector "centerOfMass" in body "arm1" of robot are mapped to the following variables:* *robot.arm1.centerOfMass[1]* *robot.arm1.centerOfMass[2]* *robot.arm1.centerOfMass[3]* *For example, a table T[4,3,2] (first dimension 4 entries, second dimension 3 entries, third dimension 2 entries) is mapped to the following Variables:* *T[1,1,1]* *T[1,1,2]* *T[1,2,1]* *T[1,2,2]* *T[1,3,1]* *T[1,3,2]* *T[2,1,1]* *T[2,1,2]* *T[2,3,1]* *…* It might occur that not all elements of an array are present. If they are present, they are given in consecutive order in the DCP slave description.

Table 9: Variable naming convention options

3.1.17.2 Outputs and Inputs

A DCP slave consumes inputs and provides outputs. Output values of a DCP slave are sent using the payload field of Data PDUs. Values of several outputs can be grouped together and sent using one Data PDU. Details are given in section 3.4.4.1.
The timing characteristics for communications are defined by the configuration of the outputs. Outputs may be sent at communication steps but must not be sent between communication steps.

Outputs with `variability = “continuous”` must be sent with their respective defined communication step size.

Outputs with `variability =”discrete”`, may be sent at every communication step size, but must be sent if the value has changed.

Discrete outputs may be mapped to continuous inputs, and vice versa.

> *Note: If a continuous output is mapped to a discrete input, zero-order-hold is implicitly introduced.*
>
> *Note: If a discrete output is mapped to a continuous input, the exact behavior might be determined by extrapolation algorithms used within the receiving DCP slave. Using such configurations, the DCP integrator and master tool should be aware of the actual behavior and subsequent effects.*

3.1.17.3 Parameters

Parameters are used to change properties of a DCP slave. They can be set by the DCP master only.

For parameters the `variability` shall be set to either `fixed` or `tunable`.

The values of parameters with `variability = “fixed”` can be set only in state `CONFIGURATION (see section 3.2.4.2)`.

The values of parameters with `variability = “tunable”` can be set at any time. The received value of a tunable parameter shall come into effect during the next computational step of a DCP slave in NRT operating mode. For the operating modes HRT and SRT the values are adopted immediately. Values of several parameters can be grouped together and sent using one Data PDU. Details are given in section 3.4.4.2.

If a value for a parameter is not set at all, it stays at its start value which is contained in the DCP slave description.

> *Note: To ensure that multiple parameters coming into effect simultaneously, they must be sent at once.*

3.1.17.4 Structural Parameters

Structural parameters may be used to indicate variable dimensions. This is used to define e.g. vectors and matrices.
Structural parameters have a start value and may be modified during simulation time.

3.1.17.5 Multidimensional Variables

An array variable is a data structure consisting of a collection of variables, each identified by an array index. A variable may have a constant number of dimensions. Each dimension has a size. A size may either be a constant or a structural parameter. Both may use a serialized start value.

The numbering of dimensions is done from left to right and from top to bottom.

> *For a C API: array[dim1][dim2]…[dimN], where N ∊ ℕ.*
>
> *For XML: document order.*

Serialization example

A = $\begin{bmatrix} a11 & a12 \\ a21 & a22 \\ a31 & a32 \end{bmatrix}$ *is serialized as follows:*

A[0][0]=a11, memory address A,

A[0][1]=a12, memory address A+1,

A[1][0]=a21, memory address A+2,

A[1][1]=a22, memory address A+3,

A[2][0]=a31, memory address A+4,

A[2][1]=a32, memory address A+5.

3.1.18 Data Type Conversions

A DCP slave shall be able to perform data type conversions for inputs and parameters as specified in Table 10. The character "x" indicates that the given conversion is allowed and feasible. For inputs and tunable parameters, an invalid conversion shall be detected in state `CONFIGURATION`. In that case, an error code shall be sent as a response to PDU `CFG_input` and PDU `CFG_tunable_parameter`.

Note: Empty cells are considered as invalid conversions.

		DCP input data types											
		uint8	uint16	uint32	uint64	int8	int16	int32	int64	float32	float64	binary	string
DCP output data types	uint8	x	x	x	x		x	x	x	x	x		
	uint16		x	x	x			x	x	x	x		
	uint32			x	x				x		x		
	uint64				x								
	int8					x	x	x	x	x	x		
	int16						x	x	x	x	x		
	int32							x	x		x		
	int64								x				
	float32									x	x		
	float64										x		
	binary											x	
	string												x

Table 10: Data type conversions

3.1.19 Native and Non-Native DCP Specification

This section defines the term *native DCP specification*. Native DCP means that the mapping of PDUs to the transport protocol preserves the bit sequence. The bit sequence of PDUs is specified in section 3.3. All PDUs, especially the Control PDUs, must be transferable via the chosen transport protocol. No additional mechanisms for exchange of information, e.g. for configuration are needed.

Note: The DCP specification for UDP/IPv4 over Ethernet follows native DCP, for example.

In contrast to the native DCP specification, the non-native DCP specification uses a different mapping to associate the DCP protocol and PDUs to a transport protocol. Available mappings are specified in section 4.

Note: The DCP specification for CAN bus follows non-native DCP, for example.

3.1.20 Permissible Payload Size

Native DCP specifies no fragmentation mechanisms. The used transport protocol must support a minimum payload size of 24 bytes. In the DCP slave description the maximumPayloadSize can be defined. It represents the maximum possible size of a data PDU in bytes. It applies to both sent and received data PDUs.

Note: The master may use CFG_input/CFG_output/CFG_tunable_parameter to ensure that the maximumPayloadSize of data PDUs is not exceeded.

3.1.21 Transport Protocol Numbering

The transport protocols supported by this DCP specification are numbered as follows.

Transport protocol	Number$_{hex}$
UDP/IPv4	0x00
rfcomm/Bluetooth	0x01
CAN based	0x02
USB (2.0)	0x03

Table 11: Transport protocol numbering

3.1.22 Logging

The DCP supports the transmission of arbitrary log data from a DCP slave to its master. For that, it defines two different approaches, namely *log on request* (LoR) and *log on notification* (LoN).
For LoR, log messages are stored within the DCP slave. They are picked up by the master on request at any time. LoR supports the delivery of multiple log messages at one time.
For LoN, log messages are not stored within the DCP slave. They are transmitted to the master immediately. LoN supports the delivery of a single log message at one time.

The exact format of a log message is defined in the DCP slave description using log templates. A DCP slave only delivers argument values to fill into this template. The full log message is then generated by the master.

Note: The length of all PDUs exchanged for logging may be precalculated using the DCP slave description.

3.1.22.1 Log Mode

A log mode for specific log messages is set by the master using the PDU CFG_logging. The default value for all log messages is No logging via DCP (0). See section 3.3.3.9 for a list of valid log modes.

3.1.22.2 Log Level

A log level is assigned to a log template in the DCP slave description. See section 3.3.3.7 for a list of valid options.

Note: This corresponds to the status field of FMI.

3.1.22.3 Log Category

A log category is both defined and assigned to a log template in the DCP slave description. One byte shall be reserved to identify a log category. See section 3.3.3.6 for a valid list of ranges.

3.2 State Machine Definitions

3.2.1 General

The state machine defined in this section is intended for use within a DCP slave. Figure 1 shows the DCP slave state machine in UML notation.

Transitions are triggered either by PDUs of the state change family (STC) or internal signals. The PDUs that trigger a transition are indicated with a STC prefix (see section 3.3). Internal signals that trigger a transition are indicated via the SIG prefix. Signals are DCP slave internal only and are therefore not exchanged via DCP PDUs. All transitions are defined in Section 3.2.5. After a transition has been performed, the slave informs the master about its new state (using the PDU NTF_state_changed)

3.2.2 Description

The state machine's entry point is labelled with `entryPoint`, whereas its exit point is labelled with `exitPoint`. If the software component implementing the DCP is not yet loaded, the DCP slave does not exist yet. After unloading the software component implementing the DCP, the DCP slave does not exist anymore.

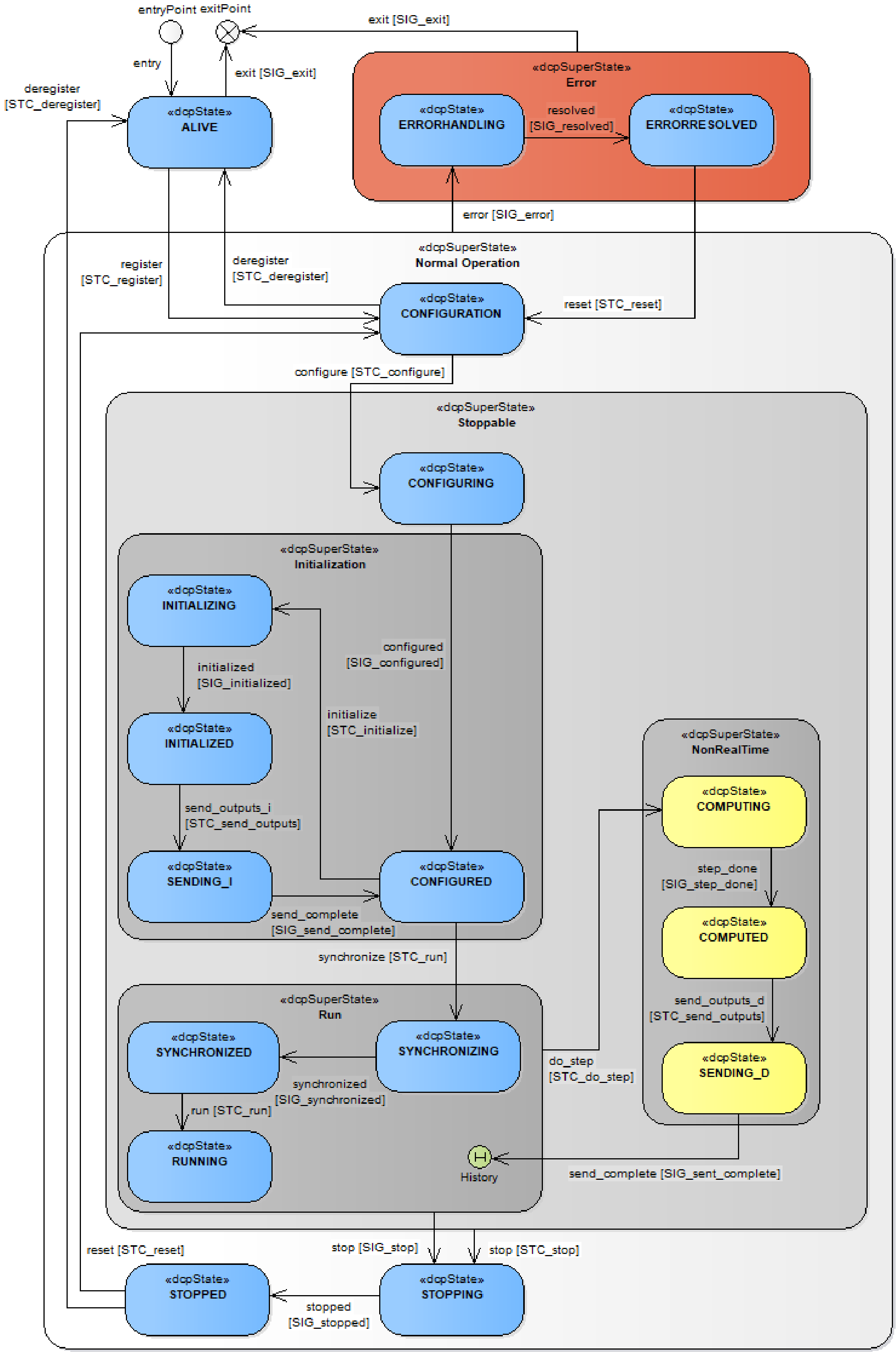


Figure 1: DCP slave state machine

3.2.3 Superstates

The following sections describe the general behavior of the defined superstates.

3.2.3.1 Normal Operation

The states CONFIGURATION, CONFIGURING, CONFIGURED, INITIALIZING, INITIALIZED, SENDING_I, RUNNING, COMPUTING, COMPUTED, SENDING_D, STOPPING, and STOPPED belong to a superstate called “Normal Operation”. This superstate assumes that the DCP slave operates as intended by the DCP slave provider.

3.2.3.2 Error

The states ERRORHANDLING and ERRORRESOLVED belong to a superstate Error. This superstate is used to handle exceptional conditions that are defined by the DCP slave provider.

3.2.3.3 Initialization

The DCP superstate Initialization is used by a DCP master to align multiple DCP slaves before running a simulation. The states CONFIGURED, INITIALIZING, INITIALIZED and SENDING_I together with their transitions allow the master to apply iterative algorithms to reach a consistent initial state over all slaves within a scenario. Initialization is independent from absolute time and the chosen operating mode.

3.2.3.4 Run

The states SYNCHRONIZING, SYNCHRONIZED and RUNNING belong to superstate Run. In contrast to superstate Initialization simulation time can elapse.

For real-time operating modes SRT and HRT simulation time is running and data is exchanged using the defined step size. For non-real-time operating mode NRT advance of simulation time and data exchange are handled as described in superstate NonRealTime.

The two states SYNCHRONIZING and RUNNING allow for distinction between a possible initial transient oscillation phase and the actual simulation experiment. By transitioning to state SYNCHRONIZED the slave indicates that it has finished the transient oscillation phase.

> *Note: For example, when the control loop between an engine test bench and a simulation model is closed, typically initial transient oscillations occur. The actual simulation experiment should only be started after this initial transient oscillation phase.*
>
> *The initial transient oscillation phase takes place in state SYNCHRONIZING. As soon as this phase is finished, the slave transitions to state SYNCHRONIZED. As soon as all slaves are in state SYNCHRONIZED, the master triggers the transition to state RUNNING. This leads to a defined point in time when the actual simulation experiment starts.*

For NRT operating mode, from each state of the superstate Run transition to state COMPUTING of superstate NonRealTime is possible. On reentry from state SENDING_D to superstate Run the entry state is the last state from which the superstate Run was left. This is indicated by the History element in the state chart (see Figure 1).

3.2.3.5 NonRealTime

The states COMPUTING, COMPUTED, and SENDING_D belong to superstate NonRealTime.

The states of NonRealTime are used for triggering calculation, advance of simulation time, and data exchange.

> *Note: (to be edited, example why three states are necessary) Consider 1 Master and 2 Slaves A and B, including Slave-to-slave communication.*
>
> *Starting point all slaves are in state RUNNING.*
>
> *Master sends PDU STC_do_step to both slaves.*
>
> *Slave A changes to state COMPUTING, calculates very fast, moves on to COMPUTED.*

If no state SENDING would exist, he would immediately send its outputs to slave B and changes to state RUNNING.

Slave B might receive this data before having received the STC_do_step from the master, due to network delay, latency, etc.

Thus, he would calculate with input data not consistent to current simulation time instance.

With a separate state SENDING the master can prevent this.

3.2.3.6 Stoppable

The states CONFIGURING, CONFIGURED, INITIALIZING, INITIALIZED, RUNNING, COMPUTING, COMPUTED, SENDING_I, SENDING_D are grouped in a super-state "Stoppable". This means that a slave is allowed to transit to the state Stopping from one of these states.

3.2.4 States

Table 12 lists the states of the state machine together with their assigned IDs.

State name	State id_{hex}
ALIVE	0x00
CONFIGURATION	0x01
CONFIGURING	0x02
CONFIGURED	0x03
INITIALIZING	0x04
INITIALIZED	0x05
SENDING_I	0x06
SYNCHRONIZING	0x07
SYNCHRONIZED	0x08
RUNNING	0x09
COMPUTING	0x0A
COMPUTED	0x0B
SENDING_D	0x0C
STOPPING	0x0D
STOPPED	0x0E
ERRORHANDLING	0x0F
ERRORRESOLVED	0x10

Table 12: State IDs

3.2.4.1 State ALIVE

General	The DCP slave is connected to communication media and waits for a DCP master to take ownership. While being in this state, the DCP slave is not assigned to a DCP master yet. A DCP master may take control of a DCP slave by sending the PDU `STC_register`. *Note: A DCP slave in this state cannot be influenced in any way, except a DCP master taking ownership.*
Preconditions	The DCP slave is off.
Allowed Actions	• Exchange of DCP control and notification PDUs • Report state

Table 13: State ALIVE

3.2.4.2 State CONFIGURATION

General	A DCP master has taken ownership of the DCP slave. In this state, the DCP slave shall accept configuration request PDUs

	(CFG). A configuration received in this state shall be applied before reaching the state CONFIGURED at the latest. A DCP master may release a DCP slave by sending the PDU STC_deregister.
Preconditions	Any configurations necessary to load the DCP slave and connect it to a given media are set.
Allowed Actions	• Exchange of DCP control and notification PDUs • Report state • Configure • Instantiate model or RT system

Table 14: State CONFIGURATION

3.2.4.3 State CONFIGURING

General	The DCP slave realizes a start condition depending on parameters, but not on input values.
Preconditions	All configurations necessary for real-time and non-realtime data exchange are set by the master or in the DCP slave description.
Allowed Actions	• Exchange of DCP control and notification PDUs • Report state • Initialize model or RT system

Table 15: State CONFIGURING

3.2.4.4 State CONFIGURED

General	At entry to this state coming from CONFIGURING, a start condition depending on parameters, but not on input values has been realized by the DCP slave. The DCP slave is ready to initialize with other DCP slaves. *Note: If node time synchronization is required (e.g. for HRT operating mode), it must have been done before leaving this state via PDU STC_run because that PDU includes a time value.*
Preconditions	Start condition is realized.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Report state • Receiving of Data PDUs • Maintain initialized condition of model or RT system

Table 16: State CONFIGURED

3.2.4.5 State INITIALIZING

General	In INITIALIZING an internal initial state of the DCP slave, which is consistent to its inputs, shall be established and the outputs shall be computed. The input values from the most recent data PDU are used for internal computation. If no inputs have been received, start values defined in DCP slave description shall be used. Simulation models: Simulation time stays at start time, simulation

	models are not computed over time, but at start time. When the DCP slave finished initializing, it issues SIG_initialized which triggers the transition to leave state INITIALIZING. If the slave fails to keep the consistent internal initial state, it must perform the transition to the superstate Error. *Note: This state refers to the FMI state "initialization mode".*
Preconditions	None.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Receiving Data PDUs • Report state • Synchronize model or RT system within scenario • Indicate end of initializing

Table 17: State INITIALIZING

3.2.4.6 State INITIALIZED

General	In INITIALIZED an internal initial state of the DCP slave, which is consistent to its inputs, is established and the outputs are available. In INITIALIZED the slave remains in its consistent internal initial state. If the slave fails to keep the consistent internal initial state, it must perform the transition to the superstate Error.
Preconditions	None.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Receiving Data PDUs • Report state • Maintain synchronized condition of model or RT system within scenario

Table 18: State INITIALIZED

3.2.4.7 State SENDING_I

General	In this state the DCP slave sends its outputs.
Preconditions	None
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Sending and receiving of Data PDUs • Report state • Indicate end of sending

Table 19: State SENDING_I

3.2.4.8 State SYNCHRONIZING

General	For real-time operating modes SRT and HRT: The DCP slave is running and inputs/outputs are exchanged. Simulation time is mapped to absolute time. For non-real-time operating mode (NRT): Simulation time is not advanced but can be increased by transitioning to the NRT-specific state COMPUTING. The DCP slave can receive inputs.

	This state is used to account for initial transient oscillations.
Preconditions	None
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Sending and receiving of Data PDUs • Report state • Indicate end of sending

Table 20: State SYCHRONIZING

3.2.4.9 State SYNCHRONIZED

General	For real-time operating modes SRT and HRT: The DCP slave is running and inputs/outputs are exchanged. Simulation time is mapped to absolute time. For non-real-time operating mode (NRT): Simulation time is not advanced but can be increased by transitioning to the NRT-specific state COMPUTING. The DCP slave can receive inputs.
Preconditions	The observed initial transient oscillations have faded out.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Sending and receiving of Data PDUs • Report state • Indicate end of sending

Table 21: State SYNCHRONIZED

3.2.4.10 State RUNNING

General	For real-time operating modes SRT and HRT: The DCP slave is running and inputs/outputs are exchanged. Simulation time is mapped to absolute time. For non-real-time operating mode (NRT): Simulation time is not advanced but can be increased by transitioning to the NRT-specific state COMPUTING. The DCP slave can receive inputs. The actual simulation experiment is executed in this state.
Preconditions	None.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Receiving of Data PDUs in NRT operating mode • Receiving and sending Data PDUs in SRT and HRT operating modes • Report state

Table 22: State RUNNING

3.2.4.11 State COMPUTING

General	In this state one computational step is performed. The values from the most recent Data PDUs are used for internal computation. The virtual simulation time is incremented by the number of steps given in the field `steps` of the PDU STC_do_`step` multiplied by resolution. *Note: This state applies to NRT (non-real-time) operat-*

	ing mode only.
Preconditions	The DCP slave is set to NRT operating mode.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Report state • Indicate end of computational step

Table 23: State COMPUTING

3.2.4.12 State COMPUTED

General	In this state all computations were performed and the DCP slave is ready to send computation results. The DCP slave can receive inputs. *Note: This state applies to NRT (non-real-time) operating mode only.*
Preconditions	The DCP slave is set to NRT operating mode.
Allowed Actions	• Exchange of DCP Control and Notification PDUs • Receiving of Data PDUs • Report state

Table 24: State COMPUTED

3.2.4.13 State SENDING_D

General	In this state the DCP slave sends its outputs. *Note: This state applies to NRT (non-real-time) operating mode only.*
Preconditions	The DCP slave is set to NRT operating mode.
Allowe Actions	• Exchange of DCP Control and Notification PDUs • Sending and receiving of Data PDUs • Report state • Indicate end of sending

Table 25: State SENDING_D

3.2.4.14 State STOPPING

General	The simulation run has finished and is now being stopped.
Preconditions	None.
Possible Actions	• Exchange of DCP Control and Notification PDUs • Report state • Indicate halt of model or RT system

Table 26: State STOPPING

3.2.4.15 State STOPPED

General	The DCP slave waits for further Control PDUs.
Preconditions	The DCP slave has come to a stop.

Possible Actions	• Exchange of DCP control and notification PDUs • Report state • Maintain condition of model or RT system

Table 27: State STOPPED

3.2.4.16 State ERRORHANDLING

General	The DCP slave tries to resolve an error.
Preconditions	A fault is detected.
Possible Actions	• Exchange of DCP control and notification PDUs • Report state • Resolve occurred error using error handling routines • Ensure safe condition of model or RT system • In case of success, transition self-reliantly to state ERRORRESOLVED. *Note: For detailed description of the DCP error handling procedure, see Section 3.4.9.*

Table 28: State ERRORHANDLING

3.2.4.17 State ERRORRESOLVED

General	The DCP slave has finished its error handling procedure and successfully mitigated the hazardous condition.
Preconditions	The DCP slave has handled the occurred error and mitigated the hazardous condition.
Possible Actions	• Exchange of DCP control and notification PDUs • Report state • Maintain safe condition of model or RT system until the DCP master either resets or terminates the DCP slave. *Note: For detailed description of the DCP error handling procedure, see Section 3.4.9.*

Table 29: State ERRORRESOLVED

3.2.5 Transitions

The following subsections describe the valid state transitions of the DCP slave state machine.

3.2.5.1 Transition entry

General	This transition marks the entry point to the state machine. The DCP software is loaded on the execution platform, therefore it transforms into a DCP slave.
Preconditions	None
Trigger	Load the DCP software.
States	• entryPoint -> ALIVE

Table 30: Transition entry

3.2.5.2 Transition exit

General	This transition marks the exit point from the state machine. The DCP software is unloaded from the execution platform.

	In case of an error, the occurred error either (1) could not be handled and the DCP software is unloaded from the execution platform, or (2) another error occurred before resetting the DCP slave.
Preconditions	None or unrecoverable error.
Trigger	SIG_exit
States	• ALIVE -> exitPoint • ERRORHANDLING -> exitPoint • ERRORRESOLVED -> exitPoint

Table 31: Transition exit

3.2.5.3 Transition register

General	A DCP master shall register a DCP slave to integrate it into a simulation scenario and use it for a simulation task.
Preconditions	The DCP slave is currently deregistered. The DCP slave received a STC_register PDU.
Trigger	STC_register
States	• ALIVE -> CONFIGURATION

Table 32: Transition register

3.2.5.4 Transition deregister

General	A DCP master deregisters a DCP slave to release it from a simulation scenario.
Preconditions	The DCP slave is registered to a DCP master.
Trigger	STC_deregister
States	• CONFIGURATION -> ALIVE • STOPPED -> ALIVE

Table 33: Transition deregister

3.2.5.5 Transition configure

General	The DCP slave has received configuration information and shall start to realize the configuration.
Preconditions	None.
Trigger	STC_configure
States	• CONFIGURATION -> CONFIGURING

Table 34: Transition configuring

3.2.5.6 Transition configured

General	The DCP slave realized a configuration.
Preconditions	None.
Trigger	SIG_configured
States	• CONFIGURING -> CONFIGURED

Table 35: Transition configured

3.2.5.7 Transition initialize

General	The DCP slave starts to establish a consistent initial state with all other connected DCP slaves.

Preconditions	None.
Trigger	STC_initialize
States	• CONFIGURED -> INITIALIZING

Table 36: Transition initialize

3.2.5.8 Transition initialized

General	The DCP slave has established a consistent initial state with other connected DCP slaves.
Preconditions	None.
Trigger	SIG_initialized
States	• INITIALIZING -> INITIALIZED

Table 37: Transition initialized

3.2.5.9 Transition run

General	This transition indicates the start of the simulation run.
Preconditions	The DCP master has determined that simulation shall start either now or at a given time.
Trigger	STC_run
States	• CONFIGURED -> RUNNING *Note: Even if the field `time` within the PDU STC_run contains a `time` > now, the DCP slave transitions immediately to state RUNNING. In state RUNNING, it waits for `time`==now and then starts the simulated time.*

Table 38: Transition run

3.2.5.10 Transition stop (STC_stop)

General	The DCP master tells the DCP slave to halt the simulation or abort the configuration or initialization phase by sending PDU STC_stop. The DCP slave proceeds to STOPPING.
Preconditions	None.
Trigger	STC_stop
States	• CONFIGURING -> STOPPING • CONFIGURED -> STOPPING • RUNNING -> STOPPING • INITIALIZING -> STOPPING • INITIALIZED -> STOPPING • SENDING_I -> STOPPING • COMPUTING -> STOPPING • COMPUTED -> STOPPING • SENDING_D -> STOPPING

Table 39: Transition stop

3.2.5.11 Transition stop (SIG_stop)

General	The DCP slave wants to stop the simulation.
Preconditions	The DCP slave raised a SIG_stop signal. *Note: A DCP slave may request simulation stop from*

	the DCP master by triggering SIG_stop. The master notices the state change of the DCP slave and reacts accordingly, e.g. may communicate STC_stop to other DCP slaves of the same scenario.
Trigger	SIG_stop
States	• RUNNING -> STOPPING

3.2.5.12 Transition do_step

General	The DCP slave starts one computational step.
Preconditions	The DCP slave is in NRT (non-real-time) operating mode.
Trigger	STC_do_step
States	• RUNNING -> COMPUTING

Table 40: Transition do_step

3.2.5.13 Transition step_done

General	The DCP slave has finished one computational step.
Preconditions	The DCP slave is in NRT (non-real-time) operating mode.
Trigger	SIG_step_done
States	• COMPUTING -> COMPUTED

Table 41: Transition step_done

3.2.5.14 Transition send_outputs

General	The DCP slave sends its computational results.
Preconditions	The DCP slave received a STC_send_outputs PDU. The DCP slave is either in NRT (non-real-time) operating mode or in Initialization superstate.
Trigger	STC_send_outputs
States	• COMPUTED -> SENDING_D • INITIALIZED -> SENDING_I

Table 42: Transition send_outputs

3.2.5.15 Transition send_complete

General	The DCP slave has finished sending its computational results.
Preconditions	The DCP slave is either in NRT (non-real-time) operating mode or in Initialization superstate.
Trigger	SIG_send_complete
States	• SENDING_D -> RUNNING • SENDING_I -> CONFIGURED

Table 43: Transition send_complete

3.2.5.16 Transition stopped

General	The DCP slave and its underlying model or real-time system has come to a halt.
Preconditions	None.
Trigger	SIG_stopped
States	• STOPPING -> STOPPED

Table 44: Transition stopped

3.2.5.17 Transition synchronize

General	The DCP slave enters the Run superstate.
Preconditions	None.
Trigger	`STC_run`
States	• `CONFIGURED -> SYNCHRONIZING`

Table 45: Transition synchronize

3.2.5.18 Transition synchronized

General	The DCP slave indicates that synchronization is finished.
Preconditions	DCP slave internal detection of synchronization.
Trigger	`SIG_synchronized`
States	• `SYNCHRONIZING -> SYNCHRONIZED`

Table 46: Transition synchronized

3.2.5.19 Transition reset

General	The DCP slave is commanded by the DCP master to go back to state CONFIGURATION. All previously configured settings are reset.
Preconditions	None.
Trigger	`STC_reset`
States	• `STOPPED -> CONFIGURATION` • `ERRORRESOLVED -> CONFIGURATION`

Table 47: Transition reset

3.2.5.20 Transition error

General	This transition represents the start of an error handling routine.
Preconditions	The DCP slave diagnoses an error.
Trigger	`SIG_error`
States	• `CONFIGURATION -> ERRORHANDLING` • `CONFIGURING -> ERRORHANDLING` • `CONFIGURED -> ERRORHANDLING` • `INITIALIZING -> ERRORHANDLING` • `INITIALIZED -> ERRORHANDLING` • `SENDING_I -> ERRORHANDLING` • `RUNNING -> ERRORHANDLING` • `COMPUTING -> ERRORHANDLING` • `COMPUTED -> ERRORHANDLING` • `SENDING_D -> ERRORHANDLING` • `STOPPING -> ERRORHANDLING` • `STOPPED -> ERRORHANDLING`

Table 48: Transition error

3.2.5.21 Transition resolved

General	The occurred error was successfully handled.
Preconditions	The DCP slave received a resolved signal.

Trigger	`SIG_resolved`
States	• `ERRORHANDLING -> ERRORRESOLVED`

Table 49: Transition resolved

3.3 PDU Definitions

3.3.1 General

Protocol Data Units (PDUs) are transmitted via abstract channels. In practice, a communication medium must be used. DCP PDUs are categorized in families. Configuration request (CFG), state change request (STC), and information request (INF) PDUs belong to the family of Request PDUs. Together with the family of response (RSP) PDUs they make up the family of Control PDUs. The families of Notification PDUs (NTF) and Data PDUs (DAT) complete the range of available PDU families.

Control PDUs are exchanged between DCP master and DCP slaves. PDUs of the families CFG, STC and INF are only sent from the DCP master to its DCP slaves and are acknowledged by the DCP slaves via RSP PDUs. Data PDUs are not acknowledged.
If the DCP master sets up a scenario where the master relays Data between DCP slaves, then also a DCP master may send and receive Data PDUs.

> *Note: Data PDUs are not acknowledged. To ensure that corruption, loss, reordering, etc. of Data PDUs is avoided, a reliable communication medium must be used.*

3.3.2 Structuring

All PDUs are structured using fields. A field is defined by its name, a DCP compliant data type, and the position of the field within the PDU, given in bytes. Table 50 provides an overview of all specified PDU fields and their corresponding data types. Concrete PDUs are distinguished by their type (field: type_id). For all PDUs, the `type_id` is available at the beginning at position zero with a length of 1 byte. A specific PDU does not contain all remaining fields, but only those required for the specific use, as can be seen in Table 55. The upcoming subsections give detailed information about each PDU.

Field	Data type specification
data_id	uint16
denominator	uint32
error_code	uint16
log_category	uint8
log_level	uint8
log_max_num	uint8
log_mode	uint8
log_template_id	uint8
log_arg_val	byte[]
log_entries	byte[]
major_version	uint8
minor_version	uint8
numerator	uint32
op_mode	uint8
parameter_vr	uint64
param_id	uint16
payload	byte[]

pdu_seq_id	uint16
pos	uint16
receiver	uint8
resp_seq_id	uint16
scope	uint8
sender	uint8
slave_uuid	unsigned byte[16]
source_data_type	uint8
source_vr	uint64
time	int64
state_id	uint8
steps	uint32
target_vr	uint64
transport_protocol	uint8
type_id	uint8

Table 50: Field data types

3.3.3 PDU Fields

3.3.3.1 Sequence Identifiers

The PDU sequence id (field: `pdu_seq_id`) is defined as a running counter, identifying PDUs within a certain amount of time. The master must maintain a `pdu_seq_id` for each slave he is connected to. The master also defines the initial value of the `pdu_seq_id` by sending `STC_register`.

> *Note: The slave can use the `pdu_seq_id` after receiving `STC_register`.*

> *Note: It is not allowed to use one `pdu_seq_id` multiple times in a row, except at natural data type overflows. The `pdu_seq_id` must always be incremented by one.*

All PDUs of the Request PDU family contain a PDU sequence identifier field. All PDUs of the Response PDU family contain a PDU response sequence identifier field (`resp_seq_id`) which contains the value of the `pdu_seq_id` of the corresponding Request PDU.

3.3.3.2 Slave Identifier

Each DCP slave within a given simulation scenario identifies itself uniquely by using a DCP `slave id`. This DCP slave `id` is assigned by the master. The DCP `id` zero (“0”) shall be reserved for the master. The two PDU fields `sender` and `receiver` use this DCP slave `id`.

In the sender field, the id of the slave that sends the PDU is given. In the receiver field, the id of the slave that shall receive the PDU is given.

3.3.3.3 Data Identifier

The field `data_id` is the unique identifier of the payload data.

3.3.3.4 Denominator

The field `denominator` holds the integer value for the denominator of the fraction that defines resolution.

3.3.3.5 Error Code

The error code is a unique identifier for defined DCP errors.

3.3.3.6 Log Category

The log category may be used by a DCP slave vendor to categorize log messages. Table 51 gives the possible options.

Log category	Definition

0	Predefined, used to address all available log categories. *Note: CFG_logging with log category "0" will affect the configuration of all categories of a slave.* *Note: A DCP slave receiving INF_log_request with log category "0" will consider all categories.*
1-255	These log categories may be specified in the DCP slave description file. Subsequently they may be used in a log template.

Table 51: Log categories

3.3.3.7 Log Level

The log level may be used in a log template in the DCP slave description file.

> *Note: This corresponds to the status field of FMI.*

Log level	Value	Definition
Fatal	0	The simulation cannot be continued. The DCP slave will transition to the error superstate. *Note: An example for this log level are several missed heartbeats, exceeding the allowed specified time out limits.*
Error	1	The current action cannot be continued. *Note: An example for this log level is a wrong UUID in `STC_register`.*
Warning	2	The current action can be continued, but simulation results could be affected. *Note: An example for this log level is a value out of bounds.*
Information	3	This log level reflects the status of a DCP slave. *Note: An example for this log level is initialization for 40% finished.*
Debug	4	This log level is intended for debug information. *Note: An example for this log level is step size for data identifier 4 set to 100.*

Table 52: Log level definitions

3.3.3.8 Log Maximum Number

This field represents the maximum number of requested log entries.

3.3.3.9 Log Mode

This field defines the mode for log functionality. Table 53 gives the available options.

Value	Definition
0	No logging via DCP
1	Log on request
2	Log on notification

Table 53: Log modes

3.3.3.10 Log Argument Values

A byte array containing the argument values as specified in a template of the DCP slave description.

3.3.3.11 Log Template Identifier

An integer value representing the template identifier, referring to a template in the DCP slave description.

3.3.3.12 Log Entries

A byte array containing one or more log entries. See section 3.3.7.29, Table 87 for details.

3.3.3.13 Major Version

The field major_version gives the major version of the DCP to be used (see section 3.1.2).

3.3.3.14 Minor Version

The field minor_version gives the minor version of the DCP to be used (see section 3.1.2).

3.3.3.15 Numerator

The field numerator holds the integer value for the numerator of the fraction that defines the resolution (see section 3.1.15).

3.3.3.16 Operating Mode

The field op_mode holds the operation mode as defined in section 3.1.14 the DCP slave must use.

3.3.3.17 Parameter Value Reference

The field parameter_vr gives the value reference of the parameter (see section 3.1.17.3).

3.3.3.18 Parameter Identifier

The field param_id gives the unique identifier of the parameter referred to in the PDU (see section 3.1.17.3).

3.3.3.19 Payload

The field payload is used to hold information that is not fixed in general but must be configured using DCP mechanisms.

3.3.3.20 Position

The field pos gives the position of a data value in the PDU Data payload field in byte (see section 3.4.4).

3.3.3.21 Scope

The field scope gives the scope of validity for a configuration of a specified PDU Data payload field as defined in section **Fehler! Verweisquelle konnte nicht gefunden werden.**.

3.3.3.22 Slave UUID

The field slave_uuid holds the universal unique identifier of a slave. It is defined as an unsigned byte array of length 16. The string representation of slave_uuid is defined according to RFC4122 [5]. It is not required that the content of the field slave_uuid follows RFC4122.

3.3.3.23 Source Data Type

The field source_data_type holds the data type of a value in a PDU Data as defined in section 3.1.9.

3.3.3.24 Source Value Reference

The field source_vr gives the value reference of the output (see section 3.1.17.2).

3.3.3.25 State Identifier

The field state_id gives the current state of the slave as defined in Table 12.

3.3.3.26 Steps

The content of the field steps depends on the chosen operating mode. In PDU STC_do_step it defines the number of computational steps the slave must perform in state COMPUTING. In PDU CFG_steps it defines the communication step size of an output for SRT and HRT operating modes (see section 3.1.14).

3.3.3.27 Target Value Reference

The field target_vr gives the value reference of the input (see section 3.1.17.2).

3.3.3.28 Transport Protocol

In the field transport_protocol the unique identifier of the transport protocol is given. Possible options are defined in section 3.1.21.

3.3.3.29 Type Identifier

In the field type_id the unique identifier of the PDU is given. An overview of all assigned type identifiers is given in section 3.3.5, the type identifier range distribution is given in section 3.3.4.

3.3.3.30 Target Time

The target_time field represents the absolute time (see section 3.1.13.1) as a 64-bit signed integer value.

3.3.4 PDU Type Identifier Range Distribution

The field type_id contains a unique number for each PDU type. For DCP, all PDU type_ids are assigned as stated in section 3.3.5. However, the following numbering scheme applies, dependent on the PDU family (also see Sections 3.1.6).

PDU group	Start	End
(not in use)	0x00	0x00
State change (STC)	0x01	0x1F
Configuration (CFG)	0x20	0x7F
Information (INF)	0x80	0xAF
Response (RSP)	0xB0	0xDF
Notification (NTF)	0xE0	0xEF
Data (DAT)	0xF0	0xFF

Table 54: PDU type identifier range distribution

3.3.5 Generic PDU Structure

The following Table 55 defines a generic PDU structure for native DCP.

Protocol Data Unit (PDU)																																		
Data (DAT)		Notification (NTF)		Control																														
				Response (RSP)					Request																									
									Information (INF)			State change (STC)									Configuration (CFG)													
DAT_parameter	DAT_input_output	NTF_log	NTF_state_changed	RSP_log_ack	RSP_error_ack	RSP_state_ack	RSP_nack	RSP_ack	INF_log	INF_error	INF_state	STC_reset	STC_stop	STC_send_outputs	STC_do_step	STC_run	STC_initialize	STC_configure	STC_deregister	STC_register	CFG_scope	CFG_logging	CFG_param_network_information	CFG_tunable_parameter	CFG_parameter	CFG_source_network_information	CFG_target_network_information	CFG_clear	CFG_output	CFG_input	CFG_steps	CFG_time_res		
0xF1	0xF0	0xE1	0xE0	0xB4	0xB3	0xB2	0xB1	0xB0	0x82	0x81	0x80	0x0A	0x09	0x08	0x07	0x05	0x04	0x03	0x02	0x01	0x2B	0x2A	0x29	0x28	0x27	0x26	0x25	0x24	0x23	0x22	0x21	0x20	type_id	DCP Fields
y	y								y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	pdu_seq_id	
				y	y	y	y	y																									resp_seq_id	
		y	y	y	y	y	y	y																									sender	
									y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	receiver	
y																							y	y									param_id	
	y																				y					y	y		y	y	y		data_id	
																								y					y	y			pos	
																														y			target_vr	
																													y				source_vr	
																								y	y					y			source_data_type	
																							y			y	y						transport_protocol	
			y			y						y	y	y	y	y	y	y	y	y													state_id	
																																y	numerator	
																																y	denominator	
															y																y		steps	
																				y													op_mode	
					y		y																										error_code	
									y													y											log_category	
																						y											log_level	
																						y											log_mode	
									y																								log_max_num	
				y																													log_entries	
		y		y																													log_template_id	
		y		y																													log_arg_val	
																								y	y								parameter_vr	
																				y													major_version	
																				y													minor_version	
y	y																								y								payload	
																					y												scope	
																				y													slave_uuid	
		y		y												y																	time	
																							y			y	y						[medium specific]	

Table 55: Generic PDU structure

3.3.6 Allowed PDUs per State

Table 56 defines the allowed PDUs per state. If a PDU is not allowed within a certain state, e.g. a RSP_nack PDU including an error_code may be sent. Alternatively, the DCP slave may also go to an ERROR state.

The following table specifies the permissible PDUs to be sent or received for each state.

PDUs	States																
	ALIVE	CONFIGURATION	CONFIGURING	CONFIGURED	INITIALIZING	INITIALIZED	SENDING_I	SYNCHRONIZING	SYNCHRONIZED	RUNNING	COMPUTING	COMPUTED	SENDING_D	STOPPING	STOPPED	ERRORHANDLING	ERRORRESOLVED
STC_register	R																
STC_deregister		R													R		
STC_configure		R															
STC_initialize				R													
STC_run				R													
STC_do_step										1							
STC_send_outputs						R						1					
STC_stop			R	R	R	R	R	R	R	R	R	R	R				
STC_reset															R		R
RSP_ack	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
RSP_nack	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
RSP_state_ack	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
RSP_error_ack																S	S
RSP_log_ack		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
NTF_state_changed	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
NTF_log		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
INF_state	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
INF_error																R	R
INF_log		R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
CFG_steps		2															
CFG_time_res		R															
CFG_input		R															
CFG_output		R															
CFG_clear		R															
CFG_target_network_information		R															
CFG_source_network_information		R															
CFG_tunable_parameter		R															
CFG_parameter		R															
CFG_param_network_information		R															
CFG_logging		R															
CFG_scope		R															
DAT_input_output				R	3	R	6	6	6	6	6	6	7	4	4	4	4
DAT_parameter				R	R	R	5	5	5	5	5	5	5	4	4	4	4

Table 56: Allowed PDUs per state

The literals in the above table have the following meaning:

Literal	Meaning
S	Sending of this PDU is allowed
R	Receiving of this PDU is allowed
X	Sending and receiving of this PDU is allowed
1	Receiving is only allowed in non-real time operating mode
2	Receiving is only allowed in real time and soft real time operating mode
3	A slave may receive `DAT_input_output` in this state, but the receiver might not consider them till it changes to `INITIALIZING` or `RUNNING`. The values from the most recent data PDU are used for internal computation. As soon as it has finished its internal computation and just before the state is left, the current output values are sent in a `DAT_input_output`.
4	Receiving of data PDUs is allowed, but the received data is not considered. *Note: Example: this might happen in case of slave to slave data transfer. The master might have sent STC_stop to a slave which receives DAT_input_output from a slave which is still in state RUNNING.*
5	A slave may receive `DAT_parameter` in this state. `DAT_parameter` must only contain parameters with variability tunable. • In NRT mode, the received values must not be considered until the DCP slave changes to `COMPUTING`. The values from the most recent data PDU are used for internal computation. • In SRT or HRT mode the values from the most recent data PDU are used for internal computation.
6	A slave may receive `DAT_input_output` in this state. • In NRT mode, the received values must not be considered until the DCP slave changes to `COMPUTING`. The values from the most recent data PDU are used for internal computation. • In SRT or HRT mode the values from the most recent data PDU are used for internal computation. Sending of `DAT_input_output` is also allowed.
7	Same as 6, excluding SRT and HRT cases. Additionally, sending of `DAT_input_output` is also allowed.

Table 57: Key for allowed PDUs per state

3.3.7 PDU Definitions

3.3.7.1 PDU STC_register

This PDU is used by a DCP master to take ownership of a given DCP slave.

The field `slave_uuid` follows the definition of section 3.3.3.22.

The fields `major_version` and `minor_version` follow the version descriptor numbering scheme of section 3.1.2.

In the field `receiver` the master sets the slave's `slave id`.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x01
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id
5	20	byte[16]	slave_uuid
21	21	uint8	op_mode
22	22	uint8	major_version
23	23	uint8	minor_version

Table 58: STC_register

3.3.7.2 PDU STC_deregister

With the PDU STC_deregister, the slave is released from the ownership of the master. It triggers the transition to state ALIVE.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x02
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 59: STC_deregister

3.3.7.3 PDU STC_configure

This PDU is used to trigger the state transition to CONFIGURING.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x03
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 60: STC_configure

3.3.7.4 PDU STC_initialize

This PDU is used to trigger the state transition to INITIALIZING.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x04
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 61: STC_initialize

3.3.7.5 PDU STC_run

After receiving STC_run, the slave transitions immediately to state SYNCHRONIZING or RUNNING, respectively.

The field `target_time` is used to schedule the start of a simulation run in HRT or SRT operating modes. It refers to absolute time.

If `target_time` >= current absolute time, a DCP slave must wait until the point in time arrives.
If `target_time` is less than the current absolute time (`target_time` < current absolute time) the DCP slave shall respond with RSP_nack, including `error_code = INVALID_START_TIME`.
If the value of `target_time` equals zero ("0"), simulation shall start immediately. In case of non-real time operation mode (NRT), `target_time` must be ignored, considering that the simulation run is controlled by STC_do_step.

CONFIGURED -> SYNCHRONIZING
When `target_time` is reached, the slave starts to exchange data and the simulation time starts to advance.

SYNCHRONIZING -> RUNNING
When `target_time` is reached, the actual simulation experiment must start.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x05
1	2	unit16	pdu_seq_id
3	3	uint8	receiver

4	4	uint8	state_id
5	12	int64	target_time

Table 62: STC_run

3.3.7.6 PDU STC_do_step

This PDU triggers the transition to COMPUTING.

It shall only be sent to DCP slaves in non-real-time (NRT) operating mode.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x07
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id
5	8	uint32	steps

Table 63: STC_do_step

3.3.7.7 PDU STC_send_outputs

This PDU triggers the transition to SENDING_I and SENDING_D and thus the transmission of calculated simulation outputs.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x08
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 64: STC_send_outputs

3.3.7.8 PDU STC_stop

This PDU triggers the transition to state STOPPING.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x09
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 65: STC_stop

3.3.7.9 PDU STC_reset

This PDU triggers the transition to state CONFIGURATION. All configuration settings received before by configuration request PDUs (CFG) are deleted.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x0A
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	state_id

Table 66: STC_reset

3.3.7.10 PDU INF_state

This PDU requests a DCP slave's current state.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x80
1	2	unit16	pdu_seq_id
3	3	uint8	receiver

Table 67: INF_state

3.3.7.11 PDU INF_error

This PDU requests a DCP slave's reported error.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x81
1	2	unit16	pdu_seq_id
3	3	uint8	receiver

Table 68: INF_error

3.3.7.12 PDU INF_log

This PDU requests the slave to send it's logging entries of the category `log_category. The number of returned logging entries is limited to` log_max_num.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x82
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	log_category
5	5	uint8	log_max_num

Table 69: INF_log

3.3.7.13 PDU CFG_time_res

This PDU requests a DCP slave to set its time resolution.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x20
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	7	uint32	numerator
8	11	uint32	denominator

Table 70: CFG_time_res

3.3.7.14 PDU CFG_steps

This PDU requests a DCP slave to set its communication step size.
The number of steps must be larger or equal than 1.

First Position [Byte]	Last Position [Byte]	Datatype	Field Name
0	0	uint8	type_id = 0x21
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	7	uint32	steps
8	9	uint16	data_id

Table 71: CFG_steps

3.3.7.15 PDU CFG_input

In order to set up slave-to-slave `DAT_input_output` PDU communication, the DCP master must inform all DCP slaves that have inputs to be received in which format they may expect `DAT_input_output` PDUs. For that purpose, `CFG_input` is used.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x22
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	data_id
6	7	uint16	pos
8	15	uint64	target_vr

16	16	uint8	source_data_type

Table 72: CFG_input

3.3.7.16 PDU CFG_output

In order to set up slave-to-slave DAT_input_output PDU communication, the master must inform all DCP slaves that have outputs to be sent to which input and at which DCP slave the value must be sent. For that purpose, the PDU CFG_output is used.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x23
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	data_id
6	7	uint16	pos
8	15	uint64	source_vr

Table 73: CFG_output

3.3.7.17 PDU CFG_clear

The DCP slave must reset all configurations set earlier by configuration request PDUs.

Note: The operating mode and the DCP slave's slave_id are not reset.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x24
1	2	unit16	pdu_seq_id
3	4	uint16	receiver

Table 74: CFG_clear

3.3.7.18 PDU CFG_target_network_information

The PDU CFG_target_network_information is used to distribute network information. It is defined by the following general structure. The complete structure depends on the used communication medium.

The character *N* denotes the total length of one specific CFG_target_network_information in bytes. It depends on the used transport protocol.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x25
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	data_id
6	6	uint8	transport_protocol
7	N-1	See chapter 4 – communication medium specific	

Table 75: CFG_target_network_information

3.3.7.19 PDU CFG_source_network_information

The message CFG_source_network_information is used to distribute network information. It is defined by the following general structure. The complete structure depends on the communication medium used.

The character *N* denotes the total length of one specific CFG_source_network_information in bytes. It depends on the used transport protocol.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x26
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	data_id
6	6	uint8	transport_protocol
7	N-1	See chapter 4 – communication medium specific	

Table 76: CFG_source_network_information

3.3.7.20 PDU CFG_parameter

The field name payload refers to the configuration information transmitted by that PDU. The character *N* denotes the total length of one specific PDU given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x27
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	11	uint64	parameter_vr
12	12	uint8	source_data_type
13	N-1	byte[N-13]	payload

Table 77: CFG_parameter

3.3.7.21 PDU CFG_tunable_parameter

This PDU is used to inform the DCP slave about the parameter format to expect.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x28
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	param_id
6	7	uint16	pos
8	15	uint64	parameter_vr
16	16	uint8	source_data_type

Table 78: CFG_tunable_parameter

3.3.7.22 PDU CFG_param_network_information

The message CFG_param_network_information is used to distribute network information. It is defined by the following general structure, the complete structure depends on the used transport protocol.

The character *N* denotes the total length of one specific CFG_param_network_information in bytes, it depends on the used transport protocol.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x29
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	param_id
6	6	uint8	transport_protocol
7	N-1	See chapter 4 - communication medium specific	

Table 79: CFG_param_network_information

3.3.7.23 PDU CFG_logging

This PDU is used to set up the DCP logging mechanisms. See section 3.1.22 for details.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x2A
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	4	uint8	log_category
5	5	uint8	log_level
6	6	uint8	log_mode

Table 80: CFG_logging

3.3.7.24 PDU CFG_scope

This PDU is used to set the scope of the configurations of a Data PDU identified by data_id. See section 3.3.3.21 and section **Fehler! Verweisquelle konnte nicht gefunden werden.** for details.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0x2B
1	2	unit16	pdu_seq_id
3	3	uint8	receiver
4	5	uint16	data_id
6	6	uint8	scope

Table 81: CFG_scope

3.3.7.25 PDU RSP_ack

This PDU is used for general acknowledgment of any requests.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xB0
1	2	uint16	resp_seq_id
3	3	unit8	sender

Table 82: RSP_ack

3.3.7.26 PDU RSP_nack

RSP_nack shall be sent whenever a received PDU MSG was not understood correctly or currently cannot be executed. It contains an error_code field indicating the error.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xB1
1	2	uint16	resp_seq_id
3	3	unit8	sender
4	5	uint16	error_code

Table 83: RSP_nack

3.3.7.27 PDU RSP_state_ack

This PDU is used by a DCP master to report the current state in the field state_id. The sender field holds the slave id of the slave. In case the slave is in state ALIVE i.e. it has not been registered and thus has not been given a slave id, the slave uses the DCP slave id from the receiver field of PDU INF_state to answer with the sender field of PDU RSP_state_ack.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xB2
1	2	uint16	resp_seq_id

3	3	unit8	sender
4	4	uint8	state_id

Table 84: RSP_state_ack

3.3.7.28 PDU RSP_error_ack

This PDU is used by a DCP slave to report the current error.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xB3
1	2	uint16	resp_seq_id
3	3	unit8	sender
4	5	uint16	error_code

Table 85: RSP_error_ack

3.3.7.29 PDU RSP_log_ack

The character *N* denotes the total length of one specific PDU given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xB4
1	2	uint16	resp_seq_id
3	3	unit8	sender
4	N-1	byte[N-4]	log_entries

Table 86: RSP_log_ack

The `log_entries` field of `RSP_log_ack` contains an array of log entries, where one single log entry has the following structure. The character *M* denotes the total length of one single log entry given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	7	int64	time
8	8	unit8	log_template_id
9	M-1	byte[M-9]	log_arg_val

Table 87: Single log entry

The `log_arg_val` field of a single log entry contains an array of variables.

3.3.7.30 PDU NTF_state_changed

This PDU indicates a successful state transition.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xE0
1	1	unit8	sender
2	2	uint8	state_id

Table 88: NTF_state_changed

3.3.7.31 PDU NTF_log

This PDU is used to send a single log entry. The payload field of NTF_log contains an array of argument values for the given `log_template_id`. The character "N" denotes the total length of one specific PDU given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xE1
1	1	unit8	sender
2	9	int64	time
10	10	uint8	log_template_id

11	N-1	byte[N-11]	log_arg_val

Table 89: NTF_log

3.3.7.32 PDU DAT_input_output

The field name payload refers to the payload of that PDU. The character "N" denotes the total length of one specific PDU given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xF0
1	2	unit16	pdu_seq_id
3	4	uint16	data_id
5	N-1	byte[N-5]	payload

Table 90: DAT_input_output

3.3.7.33 PDU DAT_parameter

The field name payload refers to the payload of that PDU. The character "N" denotes the total length of one specific PDU given in bytes. It depends on the used configuration.

First Position [Byte]	Last Position [Byte]	Data type	Field
0	0	uint8	type_id = 0xF1
1	2	unit16	pdu_seq_id
3	4	uint16	param_id
5	N-1	Byte[N-5]	payload

Table 91: DAT_parameter

3.4 Protocol

In this section the valid sequence of exchanged PDUs is defined. The following tables contain a sequence number. In this sequence number, digits define a mandatory sequence of actions, whereas letters define exclusive options.

3.4.1 Configuration Request Pattern

The DCP master may request from a DCP slave that it applies certain configuration settings by sending configuration request PDUs (PDU family CFG).

Sequence Number	Action
1	The DCP master requests a configuration setting by sending a configuration request PDU (CFG) to the DCP slave.
2a	The slave responds to the DCP master by sending RSP_nack. In that case, the DCP slave did not receive the request properly or will not be able to fulfill the request properly.
2b	The DCP slave responds by sending RSP_ack. In that case, the DCP slave will apply the desired configuration setting immediately.
2c	The DCP slave responds with RSP_nack, whenever an attempt is made to modify a configuration setting fixed in the DCP slave description. If the request is consistent with the current configuration setting, RSP_ack shall be sent.

Table 92: Configuration request pattern

3.4.2 State Transition Pattern

In order to operate the DCP state machine, the following state transition pattern is introduced. It is valid for the entire state machine, unless noted otherwise.

Sequence Number	Action

1	The DCP master requests a state transition by sending a state change request PDU to the DCP slave.
2a	The DCP slave responds to the DCP master by sending PDU RSP_nack. In that case, the DCP slave did not receive the request properly or will not be able to fulfill the request properly (see Figure 2).
2b	The DCP slave responds by sending PDU RSP_ack. In that case, the DCP slave will start the transitioning process immediately.
3	If the transition is successfully finished, the DCP slave informs the DCP master by sending a PDU NTF_state_changed (see Figure 3).

Table 93: State transition pattern procedure

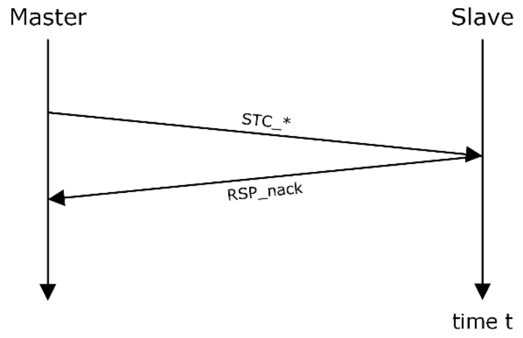


Figure 2: State transition pattern (NAck)

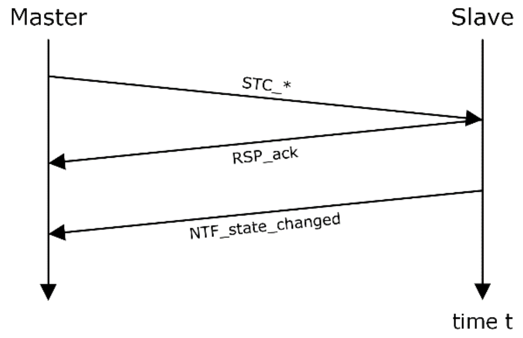


Figure 3: State transition pattern (Ack)

3.4.3 State Reporting

A DCP slave must communicate its state to its master as soon as it has changed. It does so by sending the PDU NTF_state_changed whenever a DCP slave's state change is finished.

In addition to that, the PDU INF_state can be sent at any time by the DCP master to query a DCP slave's state. A DCP slave shall respond with PDU RSP_state_ack.

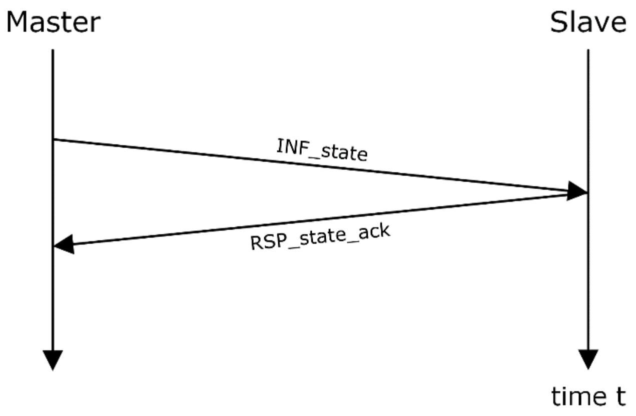


Figure 4: Positive state request

Note: The master is free to choose the DCP slave id before registering DCP slaves, to have a unique identifier for a specific DCP slave at the beginning. The slave must answer using exactly this DCP slave id.

Note: A DCP slave may be identified using the pdu_seq_id only, with the risk of hav-

ing collisions, depending on the underlying communication medium. Using the DCP slave id as described here, it is possible to uniquely identify a DCP slave.

3.4.4 Data Exchange

3.4.4.1 Inputs and Outputs

Outputs are communicated to Inputs via the payload field of PDU DAT_input_output.
The values of several outputs of one slave can be grouped in the payload field of one DAT_parameter PDU. Such a group is identified by a unique data_id. A payload field must group only values of outputs with the same configuration, i.e. sender, receiver, network configuration, scope and communication step size. The format of the payload field is defined in CONFIGURATION state using the Configuration PDUs CFG_output and CFG_input: the PDU CFG_output tells the sending slave the position of the value of an output in the payload field of a DAT_input_output.
The PDU CFG_input tells the receiving slave the position and the data type of the value in the payload field of a DAT_input_output for its input.
The communication protocol relevant information for a DAT_input_output is set in CONFIGURATION state by the PDUs CFG_set_source_network_information and CFG_set_target_network_information. For RT mode, the communication step size is set by CFG_steps.

An example of the intended sequence for the rollout of the configuration of data exchange via data objects using native DCP (UDP over IPv4) is given in the Appendix, section D.

3.4.4.2 Parameters

Parameters are set via PDU CFG_parameter.
Parameters with variability=”tunable” can additionally be set via PDU DAT_parameter. In this case the values of several parameters can be grouped in the payload field of one DAT_parameter PDU. Such a group is identified by a unique param_id. A payload field must group only values of parameters for the same configuration, i.e. receiver and network configuration. The format of the payload field is defined in CONFIGURATION state using Configuration PDUs CFG_tunable_parameter. This PDU tells the slave the position and type of the value of the parameter in the field payload of a DAT_parameter PDU.
The communication protocol relevant information for a DAT_parameter is set in CONFIGURATION state by the PDUs CFG_param_network_information.

3.4.5 Scope

Algorithms for computation of a consistent initial state (see section 3.2.3.3) may require exchange of inputs and outputs via the master. In superstates Run and NonRealTime slave-to-slave data exchange is advantageous, to reduce latencies and bandwidth compared to slave-master-slave communication. Therefore, a mechanism is defined which supports both. The PDUs CFG_scope contains the field scope. It defines in which states the respective configuration is active: Either in superstates Run and NonRealTime, or in Initialization, or both. Table 94 defines the enumeration of the field scope.

Superstates	$\text{scope}_{\text{hex}}$
Initialization/Run/NonRealTime	0x0
Initialization	0x1
Run/NonRealTime	0x2

Table 94: Enumeration of scope

3.4.6 PDU Validity

If a PDU is received by a DCP slave, it shall be checked for validity. Table 96 contains a list of all currently defined error codes which are applicable to the DCP. Table 97 contains a list of all currently defined validity checks, applicable to Control and Data PDUs. Table 98 defines the permissible actions to be taken depending on the result of PDU checking. Table 99 defines the order of error checking for Control PDUs. Table 100 defines the order of error checking for Data PDUs.

As the PDUs from the family Notification (NTF) are not intended to be received by a DCP slave, they may be dropped silently.

3.4.6.1 Error Code Ranges

The following ranges for error codes are defined.

Range	Group
0x00	NONE
0x1001-0x1FFF	PROTOCOL_ERROR_*
0x2001-0x2FFF	INVALID_*
0x3001-0x3FFF	INCOMPLETE_*
0x4001-0x4FFF	NOT_SUPPORTED_*
0x5001-0x5FFF	[Transport protocol specific error codes]
0x6001-0xFFFF	[Reserved]

Table 95: Error Code Ranges

3.4.6.2 List of Error Codes

The following list of error codes applies to the field error_code of RSP_nack and RSP_error_ack.

error_code$_{hex}$	Mnemonic	Description
0x0000	NONE	Indicates that no error is currently present.
0x1001	PROTOCOL_ERROR_GENERIC	Indicates that an error has occurred which is not specified in this document.
0x1002	PROTOCOL_ERROR_HEARTBEAT_MISSED	Indicates that the DCP slave did not receive a PDU INF_state within the maximum periodic interval t_{i_max} defined in the DCP slave description.
0x1003	PROTOCOL_ERROR_ PDU_NOT_ALLOWED_IN_THIS_STATE	Indicates that a received PDU is not allowed in the current state. See Section 3.3.6 for details.
0x1004	PROTOCOL_ERROR_PROPERTY_VIOLATED	Indicates that one of the following properties specified in the DCP slave description has been violated: min, max, preEdge, postEdge, gradient, maxConsecMissedPdus. Note: The detection and notification of a violation of these properties is optional.
0x1005	PROTOCOL_ERROR_ STATE_TRANSITION_IN_PROGRESS	Indicates that a received state change request PDU has already been acknowledged, but the current state still differs from the requested state.
0x2001	INVALID_LENGTH	Indicates that the received PDU has a valid type_id, but its length does not match.
0x2002	INVALID_LOG_CATEGORY	log_category is not log category of the slave
0x2003	INVALID_LOG_LEVEL	log_level is not a valid log level according to chapter 3.1.22.1
0x2004	INVALID_LOG_MODE	log_mode is not valid log mode according to chapter 3.1.22
0x2005	INVALID_MAJOR_VERSION	major_version as set by the master is not

		allowed according to the major version of the DCP specification defined in the DCP slave description.
0x2006	INVALID_MINOR_VERSION	minor_version as set by the master is not allowed according to the minor version of the DCP specification defined in the DCP slave description.
0x2007	INVALID_NETWORK_INFORMATION	Indicates that the network information provided through a configuration request PDU is not valid.
0x2008	INVALID_OP_MODE	Indicates that the operating mode requested through STC_register is not supported by this DCP slave.
0x2009	INVALID_PAYLOAD	Indicates that the length of a string is not equal to a defined fixedLengthBytes or the length of a binary value is greater than a defined maxSize.
0x200A	INVALID_SCOPE	scope is invalid according to chapter **Fehler! Verweisquelle konnte nicht gefunden werden.**
0x200B	INVALID_SOURCE_DATA_TYPE	source_data_type is not compatible with the inputs data type. For a list of data types see section 3.1.9, and for casting rules see section 3.1.18.
0x200C	INVALID_START_TIME	Indicates that the start time provided in STC_run is invalid, e.g. in the past.
0x200D	INVALID_STATE_ID	state_id is not equal to the slave's state.
0x200E	INVALID_STEPS	Indicates that the number of steps in CFG_steps is not supported.
0x200F	INVALID_TIME_RESOLUTION	Indicates that the time resolution expressed by numerator and denominator is not valid.
0x2010	INVALID_TRANSPORT_PROTOCOL	The given transport_protocol is not supported by the slave
0x2011	INVALID_UUID	slave_uuid is not equal to slave's uuid.
0x2012	INVALID_VALUE_REFERENCE	Indicates that the value reference in CFG_input, CFG_output, CFG_parameter, or CFG_tunable_parameter is not available within that DCP slave.

0x3001	INCOMPLETE_CONFIG_GAP_INPUT_POS	There are gaps in the configured PDU Data payload field. Note: No gap means if pos n is not the last pos, there exists a pos n+1.
0x3002	INCOMPLETE_CONFIG_GAP_OUTPUT_POS	There are gaps in the pos of the received CFG_output PDUs.
0x3003	INCOMPLETE_CONFIG_GAP_TUNABLE_POS	There are gaps in the pos of the received CFG_tunable_parameter PDUs.
0x3004	INCOMPLETE_CONFIG_NW_INFO_INPUT	For the PDU Data payload field identified by data_id no or no valid CFG_source_network_information has been received.
0x3005	INCOMPLETE_CONFIG_NW_INFO_OUTPUT	For the PDU Data payload field identified by data_id no or no valid CFG_target_network_information has been received.
0x3006	INCOMPLETE_CONFIG_NW_INFO_TUNABLE	Not for each param_id which occurs in the PDUs CFG_tunable_parameter a valid

		CFG_param_network_information with the same param_id is received.
0x3007	INCOMPLETE_CONFIG_SCOPE	At least one data_id is missing the setting of the scope.
0x3008	INCOMPLETE_CONFIG_STEPS	For every data_id which occurs in a CFG_output at least one valid CFG_steps with the same data_id must have been received.
0x3009	INCOMPLETE_CONFIG_TIME_RESOLUTION	No time resolution is specified. Neither through the DCP slave description, nor through CFG_time_res.
0x300A	INCOMPLETE_CONFIGURATION	Indicates that a DCP slave cannot leave CONFIGURATION state due to missing configuration information.

0x4001	NOT_SUPPORTED_LOG_ON_NOTIFICATION	log_mode logOnNotification is not supported by the slave, i.e. in DCP slave description canProvideLogOnNotification = false
0x4002	NOT_SUPPORTED_LOG_ON_REQUEST	log_mode logOnRequest is not supported by the slave i.e. in DCP slave description canProvideLogOnRequest = false
0x4003	NOT_SUPPORTED_VARIABLE_STEPS	Steps differ from previous ones (if the slave does not support variable step sizes).
0x4004	NOT_SUPPORTED_TRANSPORT_PROTOCOL	Indicates that the transport protocol number provided through a configuration request PDU is not supported.
0x4005	NOT_SUPPORTED_PDU	Indicates that this type of PDU (identified through field type_id) is defined within this specification but is not supported by this DCP slave. *This affects the current operating mode as well as DCP slave capabilities.*

Table 96: List of Error Codes

Check	Details
Type identifier	The type_id field of the received PDU is checked. All valid type identifiers are specified in Section 3.3. *Note: This affects type_identifiers which are not specified, as well as PDUs which are not intended to be processed by this DCP slave, e.g. RSP_ack.*
Length	The PDU length in bytes is checked. *Note: A plausibility check would include checking for PDU length smaller than 4 Bytes, resulting in an immediate drop. Detailed length checks shall be performed as defined in Table 99.*
Support	The received PDU is supported by the DCP slave, e.g. capability flag canSupportReset is set.
Receiver	The receiver field of the received PDU is evaluated against the assigned DCP slave id. In state ALIVE, check if receiver is greater than zero. *Note: The DCP slave id zero is reserved for the master. Therefore no slave can be a valid receiver of a PDU which is determined for receiver zero.*
Sequence id	The pdu_seq_id field of the received PDU is evaluated and checked for integrity. *Note: For integrity checking of the pdu_seq_id the approach highlighted in the Appendix may be used.* The maxConsecMissedPdus attribute from DCP slave description may influence the exact behavior.
data_id/param_id	The data_id or param_id field is validated.
State	The type_id field of the received PDU is validated against the current DCP slave state according to Table 56.
Semantics	All fields of the received PDU, that are specified in Section 3.3 and are not explicitly covered in this section, shall be checked for validity and integrity.

Table 97: Applicable PDU validity checks

Activity	Details
Drop PDU	The received PDU shall be dropped silently, without any further actions.
Process PDU	The received PDU shall be processed further according to this specification document.
Handle error	The occurred error shall be communicated to the DCP master. Possible actions include transition to an Error state, and responding an error_code by using RSP_nack or RSP_error_ack. A list of corresponding error_codes can be found in Section 3.4.6.3.

Table 98: Applicable actions related to PDU validity checks

3.4.6.3 Order of Error Checking

The following defines the order of the checks for PDUs a DCP slave must perform. If multiple checks fail, the action from the failed check with the lowest order value must be performed first. The error codes apply to the fields error_code of RSP_nack and RSP_error_ack.

NOTE: This does not necessarily apply to the order of internally performed checks. This order indicates the sequence of error reporting. This is independent of the sequence of actual performed checks.

Order	Check	Action/Error code
1	Type identifier	Drop
	Receiver	
	Sequence ID	
2	Support	UNSUPPORTED_PDU
3	Length	INVALID_LENGTH
4	State	PDU_NOT_ALLOWED_IN_THIS_STATE
5	Semantics	See section 3.4.6.4

Table 99: Order of Error Checking for Control PDUs

Order	Check	Action
1	Type identifier	Drop
	data_id/param_id	
	Sequence ID	
	Length	
	State	

Table 100: Order of Error Checking for Data PDUs

3.4.6.4 Order of Error Codes

Order	Error Code
1	INVALID_STATE_ID
2	INVALID_UUID
3	INVALID_OP_MODE
4	INVALID_MAJOR_VERSION
5	INVALID_MINOR_VERSION

Table 101: Error code order for STC_register

Order	Error Code
1	INVALID_STATE_ID

Table 102: Error code order for STC_deregister

Order	Error Code if condition fails
1	INVALID_STATE
2	INCOMPLETE_CONFIG_GAP_INPUT_POS
3	INCOMPLETE_CONFIG_GAP_OUTPUT_POS
4	INCOMPLETE_CONFIG_GAP_TUNABLE_POS
5	INCOMPLETE_CONFIG_NW_INFO_INPUT
6	INCOMPLETE_CONFIG_NW_INFO_OUTPUT
7	INCOMPLETE_CONFIG_NW_INFO_TUNABLE
8	INCOMPLETE_CONFIG_STEPS
9	INCOMPLETE_CONFIG_TIME_RESOLUTION
10	INCOMPLETE_CONFIG_SCOPE

Table 103: Error code order for STC_configure

Order	Error Code if condition fails
1	INVALID_STATE_ID

Table 104: Error code order for STC_initialize

Order	Error Code if condition fails
1	INVALID_STATE_ID
1	INVALID_START_TIME

Table 105: Error code order for STC_run

Order	Error Code if condition fails
1	INVALID_STATE_ID
2	INVALID_STEPS
3	NOT_SUPPORTED_VARIABLE_STEPS

Table 106: Error code order for STC_do_step

Order	Error Code if condition fails
1	INVALID_STATE_ID

Table 107: Error code order for STC_send_outputs

Order	Error Code if condition fails
1	INVALID_STATE_ID

Table 108: Error code order for STC_stop

Order	Error Code if condition fails
1	INVALID_STATE_ID

Table 109: Error code order for STC_reset

NOTE: A received STC_reset PDU is caught by PDU support check if canHandleReset = false (see Table 99)

Order	Error Code if condition fails
1	INVALID_LOG_CATEGORY

Table 110: Error code order for INF_log

Note: A received configuration PDU is caught by Check PDU support if canAcceptConfigPdus = false.

Order	Error Code if condition fails
1	INVALID_TIME_RESOLUTION

Table 111: Error code order for CFG_time_res

Order	Error Code if condition fails
1	INVALID_STEPS

Table 112: Error code order for CFG_steps

Order	Error Code if condition fails
1	INVALID_VALUE_REFERENCE
2	INVALID_SOURCE_DATA_TYPE

Table 113: Error code order for CFG_input

Order	Error Code if condition fails
1	INVALID_VALUE_REFERENCE
2	INVALID_STEPS

Table 114: Error code order for CFG_output

Order	Error Code if condition fails
1	INVALID_TRANSPORT_PROTOCOL
2	INVALID_NETWORK_INFORMATION

Table 115: Error code order for CFG_target_network_information

Order	Error Code if condition fails
1	INVALID_TRANSPORT_PROTOCOL
2	INVALID_NETWORK_INFORMATION

Table 116: PDU Error code order CFG_source_network_information

Order	Error Code if condition fails
1	INVALID_VALUE_REFERENCE
2	INVALID_SOURCE_DATA_TYPE
3	INVALID_PAYLOAD

Table 117: Error code order for CFG_parameter

Order	Error Code if condition fails
1	INVALID_VALUE_REFERENCE
2	INVALID_SOURCE_DATA_TYPE

Table 118: PDU Error code order CFG_tunable_parameter

Order	Error Code if condition fails
1	INVALID_TRANSPORT_PROTOCOL
2	(Driver specific error handling.)

Table 119: Error code order for CFG_param_network_information

Note: A received CFG_logging is caught by PDU support check if canProvideLogOnRequest = false and canProvideLogOnNotification = false (see Table 99).

Order	Error Code if condition fails
1	NOT_SUPPORTED_LOG_ON_REQUEST
2	NOT_SUPPORTED_LOG_ON_NOTIFICATION
3	INVALID_LOG_CATEGORY
4	INVALID_LOG_LEVEL
5	INVALID_LOG_MODE

Table 120: Error code order for CFG_logging

Order	Error Code if condition fails
1	INVALID_SCOPE

Table 121: Error code order for CFG_scope

3.4.7 Error Reporting

Whenever a DCP slave is in ERRORHANDLING or ERRORRESOLVED states, the DCP master may send INF_error to this DCP slave, to find out about the reason.

Sequence Number	Action
1	The DCP master sends INF_error to the DCP slave.
2a	The DCP slave responds by sending RSP_error_ack, which contains an error code.
2b	The DCP slave responds by sending RSP_nack, if he is currently not in the Error superstate.

3.4.8 Heartbeat

The heartbeat functionality is optional. Its availability is indicated by capability flag canMonitorHeartbeat, see section 5.11.

A slave should be able to detect that its master is still active. Therefore, the master shall send a periodic PDU INF_state at a pre-defined interval t_i to each of the connected slaves. This interval is specified by the master. This enables two monitoring functions, defined as follows.

3.4.8.1 Slave Monitoring

The master receives a PDU RSP_state_ack from each slave and determines the response time. If the response time t_r exceeds a given time interval t_{r_max}, the master should take appropriate action.

3.4.8.2 Master Monitoring

Each slave must respond with a PDU RSP_state_ack immediately. A timeout defined in the DCP slave description determines the maximum waiting time t_{i_max} between two PDUs INF_state. If the timeout expires, the slave shall go to state ERRORHANDLING and subsequently to state ERRORRESOLVED.

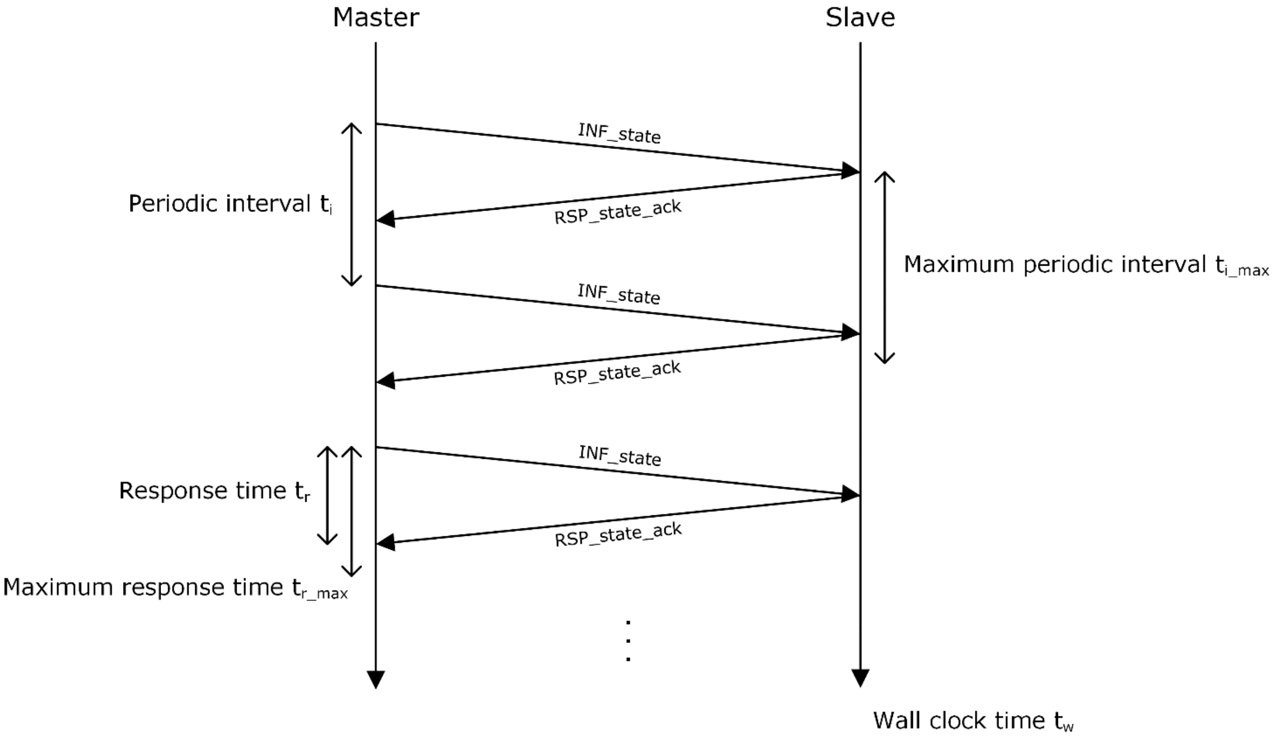


Figure 5: Heartbeat functionality

3.4.9 Error Handling

The DCP represents a framework to handle faults and errors to avoid failures of simulation scenarios, when running in SRT or HRT operation mode. Goal is to protect physical equipment (connected real-time systems) as well as human operators from any harm that may be caused during normal operation. The error handling procedures described here are in-line with ISO 26262 [6].

3.4.9.1 Unavailable communication medium

This description assumes that the used communication medium has become unavailable unexpectedly. In case a DCP slave is unable to send or receive PDUs, it transitions to ERRORHANDLING state, and subsequently to the ERRORRESOLVED state. If the communication medium becomes available again, it may react to e.g. an STC_reset. Otherwise, operator intervention is required and the DCP slave must be restarted by other means.

3.4.9.2 Available communication medium

In the following the procedure of error handling is described under the assumption that it is still possible to exchange PDUs.

Sequence Number	Action
1	A fault (faulty behavior or condition in model or RT system) occurs within a DCP slave (time t_f). After the DCP slave detected this fault (time t_{fd}) it transitions self-reliantly to the Error superstate. *Note: The transition to ERRORHANDLING is not requested from the master.*
2	The DCP slave transitions to the ERRORHANDLING state immediately.
3	Within the ERRORHANDLING state, the DCP slave tries to send an NTF_state_changed to the master. Then it starts suitable error handling routines and tries to resolve the error. *Note: Appropriate measures of error resolving are shutdown of*

	subsystems, potential energy dissipation in connected RT systems, etc. This may take some time
4a	If successful, the DCP slave transitions to ERRORRESOLVED state immediately. *Note: No request to do so from master.* The DCP slave either sends a PDU NTF_state_changed to its master, reporting that the state transition is finished, or reports the state change to the master on request. *Note: As defined in communication pattern.*
4b	If not successful, the system experiences an unrecoverable error. The transition to state ERRORRESOLVED is not performed. Signal SIG_exit to shut down in terms of an error handling procedure may be called.
5	In state ERRORRESOLVED, the DCP slave receives a PDU STC_reset (time t_r). It acknowledges it by sending a PDU RSP_ack to the DCP master.
6	The transition to state CONFIGURATION, of superstate Normal operation is performed. Either a PDU NTF_state_changed is sent to the master or the state change is reported to the master on request.

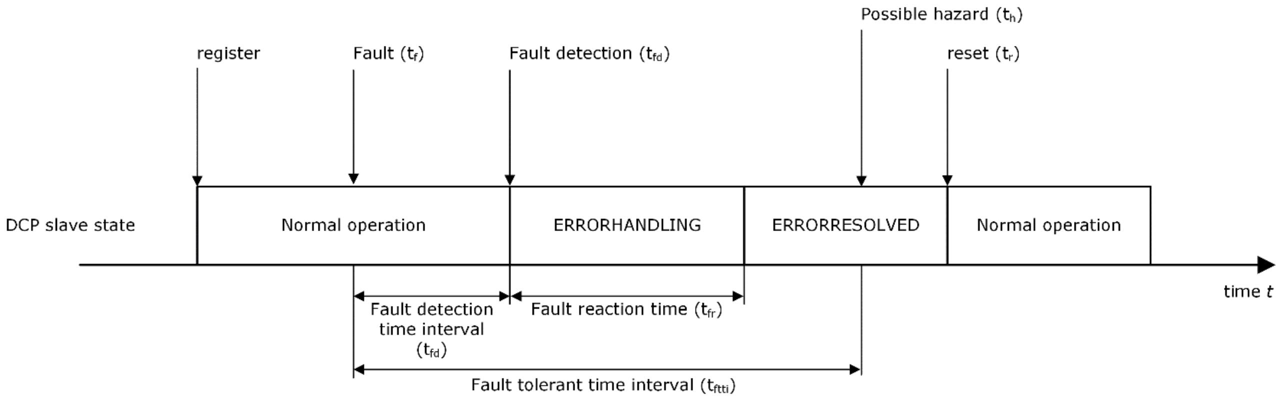


Figure 6: Procedure for error handling

3.4.10 Unintended Behaviour

If a DCP slave receives a valid, previously received PDU but with a different pdu_seq_id number, it shall acknowledge and process it as specified.

4 Transport Protocols

4.1 General

This section specifies the underlying transport protocols for DCP. Furthermore, this section assumes the default DCP slave integration as given in Figure 32 on page 98. Therefore, the DCP integrator must know transport protocol specific information to connect the provided DCP slaves to a communication medium. For this step, the DCP slave description file may be helpful.

4.2 User Datagram Protocol (UDP/IPv4)

4.2.1 General

A DCP slave using UDP/IPv4 is accessible through an IP address and a port number. This information is defined within the DCP slave description. For communication with a DCP master, the DCP slave replies to the IP address and port where the initial `STC_register` is coming from.

> *Note: A DCP master could be implemented as a UDP/IPv4 client, whereas a DCP slave could be implemented as a UDP/IPv4 server.*

As soon as `STC_register` is received and positively acknowledged (RSP_ack sent by DCP slave and received by DCP master) the IP address and port for communication with the master are fixed for this simulation run.
The port information at the slave side is cleared, as soon as `STC_deregister` is received.

> *Note: The DCP slave sender port may differ from the DCP slave receiving port.*

4.2.2 Transport Protocol Specific Fields

4.2.2.1 Port

The field `port` specifies a UDP port number.

4.2.2.2 IP Address

The field `ip_address` specifies an internet protocol address.

4.2.3 Network Information

The master distributes the communication information relevant for data exchange using the following two PDUs: `CFG_target_network_information` is sent to the sending DCP slave. It contains IP address and port number of the receiving DCP slave.

> *Implementation hint: If multiple network information with the same `data_id` is received by a DCP slave, the `DAT_input_output` PDU is to be sent to all specified targets within the `CFG_target_network_information`.*

`CFG_source_network_information` is sent to the receiving DCP slave. It contains a port number. The receiving DCP slave waits for incoming data on this port number.

> *Implementation hint: The sending port may be chosen randomly by the UDP/IP stack implementation. It is never checked in any way on the receiver side.*

The following three tables complete the PDU descriptions of sections 3.3.7.18, 3.3.7.19, and 3.3.7.22, respectively, for UDP/IPv4.

The UDP/IPv4 specific part of `CFG_target_network_information` is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x0
7	8	uint16	port
9	12	uint32	ip_address

Table 122: CFG_target_network_information

The UDP/IPv4 specific part of the corresponding CFG_param_network_information is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x0
7	8	uint16	port
9	12	uint32	ip_address

Table 123: CFG_param_network_information

The UDP/IPv4 specific part of CFG_source_network_information is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x0
7	8	uint16	port
9	12	Uint32	ip_address

Table 124: CFG_source_network_information

4.2.4 Port information

If a port is specified inside the Control element (see section 5.10.1), Control PDUs may only be received over this port.

> *Note: If no such port is specified, this can be interpreted in two different ways. First, the integrator must obtain and set this information in another way, which is not specified in this document. Second, the DCP slave is not meant to be controlled via UDP over IPv4. The DCP slave provider might be consulted for clarification.*

If a port or port range is specified inside the DAT_input_output (see section 5.10.1) element, PDUs DAT_input_output may only be received over these ports.

> *Note: If no port or port range is specified inside the DAT_input_output element, this can be interpreted in three different ways. First, the integrator must obtain and set this information in another way, which is not specified in this document. Second, the DCP master is free to choose the port numbers. The DCP slave may reject the use of requested ports using PDU RSP_nack and the corresponding error code. Third, the DCP slave is not meant to exchange PDUs DAT_input_output using UDP via IPv4. The DCP slave provider might be consulted for clarification.*

If a port or port range is specified inside the DAT_parameter element, PDUs DAT_parameter may only be received over these ports.

> *Note: If no port or port range is specified inside the DAT_parameter element, this can be interpreted in three different ways. First, the integrator must obtain and set this information in another way, which is not specified in this document. Second, the DCP master is free to choose the port numbers. The DCP slave may reject the use of requested ports using PDU RSP_nack PDU and the corresponding error code. Third, the DCP slave is not meant to exchange PDUs DAT_parameter using UDP via IPv4. The DCP slave provider might be consulted for clarification.*

Some port numbers might already be used or reserved for dedicated services.

> *Note: Implementation hint: If the master is free to choose an UDP/IPv4 port, he should use the free user ports specified by IANA organization. Otherwise it is possible that the chosen port collides with one of the ports reserved by IANA.*
>
> *https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml*

4.2.5 Host information

If a host is specified inside the Control element (see section 5.10.1) Control PDUs may only be received on this host.

> *Note: If no host is specified, this can be interpreted as follows. The integrator must obtain and set this information in another way, which is not specified in this document.*

If a host is specified inside the `DAT_input_output` element PDUs `DAT_input_output` may only be received on this host.

> *Note: If no host is specified, this can be interpreted as follows. The integrator must obtain and set this information in another way, which is not specified in this document.*

If a host is specified inside the `DAT_parameter` element PDUs `DAT_parameter` may only be received on this host.

> *Note: If no host is specified, this can be interpreted as follows. The integrator must obtain and set this information in another way, which is not specified in this document.*

4.3 Bluetooth (IEEE 802.15)

4.3.1 General

A DCP slave using Bluetooth is accessible through an address (BD_ADDR) and a port number. This information is defined within the DCP slave description. For communication with a DCP master, the DCP slave replies to the address and port where the initial `STC_register` is coming from.

> *Note: A DCP master could be implemented as a Bluetooth client, whereas a DCP slave could be implemented as a Bluetooth server.*

The BD_ADDR is a unique and permanent 48-bit address number created in accordance with section 8.2 ("Universal addresses") of the IEEE 802-2014.

4.3.2 Transport Protocol Specific Fields

4.3.2.1 Port

The field `port` specifies a port number.

4.3.2.2 Bluetooth Device Address

The field `bd_addr` specifies a Bluetooth device address.

4.3.2.3 IP Address

The field `ip_address` specifies an internet protocol address.

4.3.3 Network Information

The master distributes the communication information relevant for data exchange using the following two PDUs: `CFG_target_network_information` is sent to the sending DCP slave. It contains the BD_ADDR and port number of the receiving DCP slave.

> *Note: If multiple network information with the same `data_id` is received by a DCP slave, the `DAT_input_output` PDU is to be sent to all specified targets within the `CFG_target_network_information`.*

`CFG_source_network_information` is sent to the receiving DCP slave. It contains a port number. The receiving DCP slave listens on this port number.

The Bluetooth specific part of `CFG_target_network_information` is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x1
7	7	uint8	port
8	15	uint64	bd_addr

Table 125: CFG_target_network_information

The Bluetooth specific part of the corresponding `CFG_param_network_information` is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x1
7	8	uint16	port
9	12	uint32	ip_address

Table 126: CFG_param_network_information

The Bluetooth specific part of `CFG_source_network_information` is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 0x1
7	7	uint8	port
8	15	uint64	bd_addr

Table 127: CFG_source_network_information

Note: Application hint: The RFCOMM available ports are limited to a range between 1 and 30. Some ports might be already used by the operating system

4.4 Universal Serial Bus (USB)

4.4.1 USB Version

The target USB version is 2.0. For simplicity the term USB is used for USB 2.0.

4.4.2 General

A DCP master must be implemented on the USB host side, whereas a DCP slave must be implemented as a USB device. A DCP slave using USB is accessible through the slave `uuid`. The USB host driver for DCP must manage the mapping between slave `uuid` & DCP slave id and assigned USB number.

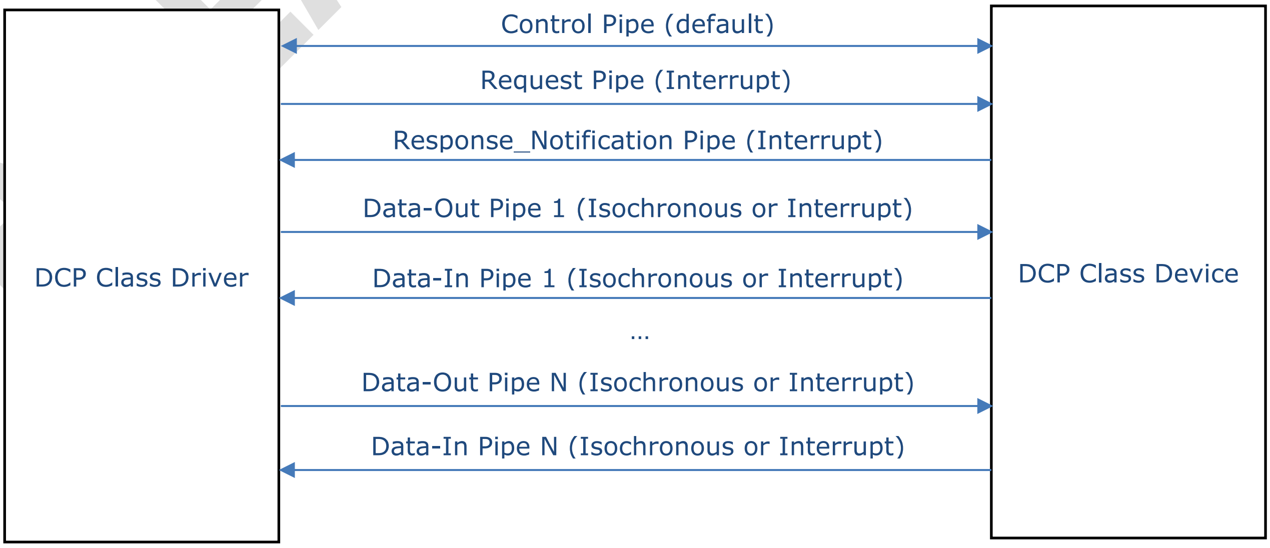

DCP Class Driver
Control Pipe (default)
Request Pipe (Interrupt)
Response_Notification Pipe (Interrupt)
Data-Out Pipe 1 (Isochronous or Interrupt)
Data-In Pipe 1 (Isochronous or Interrupt)
…
Data-Out Pipe N (Isochronous or Interrupt)
Data-In Pipe N (Isochronous or Interrupt)
DCP Class Device

Figure 7: USB scheme

The DCP protocol for USB is defined by the DCP class. Every DCP class device consists of the following pipes:

- The Control Pipe is used for receiving and responding to requests for USB control and class data.
- The Request Pipe is used for receiving Request PDUs from the master.
- The Response_Notification Pipe is used for sending Response and Notification PDUs to the master.
- A Data-Out pipe is used to receive Data from other slaves or from the master.
- A Data-In pipe is used to send `DAT_input_output` to other slaves or to the master.
- A vendor can define multiple Data-Out and Data-In pipes. There must be at least one Data-In pipe if there exists at least one output variable at the DCP slave. There must be at least one Data-Out pipe if there exists at least one input variable or parameter at the DCP slave.

4.4.3 Transport Protocol Specific PDU Fields

4.4.3.1 Endpoint Address

The field `endpoint_address` is used to specify an endpoint address.

4.4.4 Descriptors

The following chapters describe the applied USB Descriptors used for the DCP USB class.

4.4.4.1 Device

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	18
1	1	uint8	bDescriptorType	1
2	3	uint16	bcdUSB	Vendor Specific
4	4	uint8	bDeviceClass	0
5	5	uint8	bDeviceSubClass	0
6	6	uint8	bDeviceProtocol	0
7	7	uint8	bMaxPacketSize	64
8	9	uint16	idVendor	Vendor Specific
10	11	uint16	idProduct	Vendor Specific
12	13	uint16	bcdDevice	Vendor Specific
14	14	uint8	iManufacturer	Vendor Specific
15	15	uint8	iProduct	Vendor Specific
16	16	uint8	iSerialNumber	Vendor Specific[1]
17	17	unit8	bNumConfigurations	Vendor Specific

Table 128: Device descriptor

[1] The string iSerialNumber is pointing to must be equal to the slave uuid in Unicode coding.

4.4.4.2 Configuration

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	8
1	1	uint8	bDescriptorType	2
2	3	uint16	wTotalLength	Vendor Specific
4	4	uint8	bNumInterfaces	Vendor Specific
5	5	uint8	bConfigurationValue	0
6	6	uint8	iConfiguration	0

7	7	uint8	bmAttributes	Vendor Specific
8	8	uint8	bMaxPower	Vendor Specific

Table 129: Configuration descriptor

4.4.4.3 Interface

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	9
1	1	uint8	bDescriptorType	4
2	2	uint8	bInterfaceNumber	0
3	3	uint8	bAlternateSetting	0
4	4	uint8	bNumEndpoints	5
5	5	uint8	bInterfaceClass	255 [2]
6	6	uint8	bInterfaceSubClass	205 [2]
7	7	uint8	bInterfaceProtocol	205 [2]
8	8	uint8	iInterface	0

Table 130: Interface descriptor

[2] *Implementation Hint: Because an official USB class for DCP does not exist at the moment, the vendor specific class is used. Therefore 205 is selected as an arbitrary number to define the subclass and protocol.*

4.4.4.4 Endpoint

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	7
1	1	uint8	bDescriptorType	5
2	2	uint8	bEndpointAddress	16 (00010000_{Bin})
3	3	uint8	bmAttributes	3 (00000011_{Bin})
4	5	uint16	wMaxPacketSize	1024
6	6	uint8	bInterval	16

Table 131: Request Pipe

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	7
1	1	uint8	bDescriptorType	5
2	2	uint8	bEndpointAddress	33 (00100001_{Bin})
3	3	uint8	bmAttributes	3 (00000011_{Bin})
4	5	uint16	wMaxPacketSize	1024
6	6	uint8	bInterval	16

Table 132: Response_Notification Pipe

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	7
1	1	uint8	bDescriptorType	5
2	2	uint8	bEndpointAddress	Vendor Specific [3]
3	3	uint8	bmAttributes	Vendor Specific [4]
4	5	uint16	wMaxPacketSize	1024
6	6	uint8	bInterval	Vendor Specific

Table 133: Data-Out Pipe

[3] Must be constructed according to the USB standard. The endpoint number must be greater than 2. The Direction must be 0 (Out).
[4] Must be constructed according to the USB standard. The transfer type must be Isochronous or Interrupt.

First Position [Byte]	Last Position [Byte]	Data-type	Field Name	Value
0	0	uint8	bLength	7
1	1	uint8	bDescriptorType	5
2	2	uint8	bEndpointAddress	Vendor Specific [5]
3	3	uint8	bmAttributes	Vendor Specific [6]
4	5	uint16	wMaxPacketSize	1024
6	6	uint8	bInterval	Vendor Specific

Table 134: Data-In Pipe

[5] Must be constructed according to the USB standard. The endpoint number must be greater than 2. The Direction must be 1 (In).
[6] Must be constructed according to the USB standard. The transfer type must be Isochronous or Interrupt.

4.4.5 Network Information

The USB specific part of CFG_target_network_information is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 5
7	7	uint8	endpoint_address
8	23	byte[16]	slave_uuid

Table 135: CFG_target_network_information

The USB specific part of CFG_source_network_information is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 5
7	7	uint8	endpoint_address

Table 136: CFG_source_network_information

The USB specific part of CFG_param_network_information is defined by the following structure:

First Position [Byte]	Last Position [Byte]	Data type	Field
6	6	uint8	transport_protocol = 5
7	7	uint8	endpoint_address

Table 137: CFG_param_network_information

4.4.6 DAT_input_output forwarding

According to the USB standard communication is only possible between USB host and USB device. The USB host driver must forward the DAT_input_output PDU to the corresponding USB device for slave-to-slave communication.

4.5 CAN Bus Communication Systems

This specification supports CAN bus communication systems. Due to the facts that

- the CAN payload is limited to 8 bytes,
- CAN does not support fragmentation,
- CAN uses its own addressing schema (arbitration)
- and thus not all Control PDUs can be sent via CAN as defined in native DCP specification

the DCP specification for CAN bus is non-native. This specification of DCP over CAN supports the KCD file format[1]

In order to map the DCP onto the CAN bus, two additional resources are provided together with this specification:

- DCP over CAN XSD schema description (`DCP_over_CAN.xsd`)
- DCP over CAN XSL style sheet (`DCP_over_CAN_to_KCD.xsl`)

4.5.1 Procedure

The intended procedure is to use the XSD schema to create a configuration in XML file format. An XSL style sheet is then applied to this XML file, generating a valid KCD file.

4.5.2 DCP over CAN

Figure 8 shows the DCP over CAN root element.

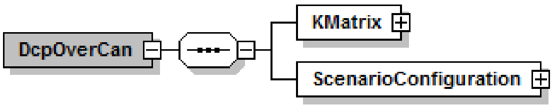


Figure 8: DcpOverCAN root element

Element name	Description
KMatrix	Contains all elements to describe the messages & signals of the CAN bus and the participation of the bus members to the messages.
ScenarioConfiguration	Contains all elements to describe the co-simulation scenario, which would be distributed over configuration PDUs in a native DCP transport protocol for each DCP slave. In addition it contains the name, DCP id & uuid. Which element belongs to which DCP slave can be determined using the uuid.

Table 138: DcpOverCan element

4.5.3 Definition of KMatrix

The element `KMatrix` is specified as follows. The KMatrix element consist optional of the CAN message description all state change, information, notification & response PDUs , as well as an arbitrary number of CAN messages for DAT_input_output & DAT_parameter PDUs. Each state change, information, notification or response PDU is defined in the following way.

[1] https://github.com/dschanoeh/Kayak

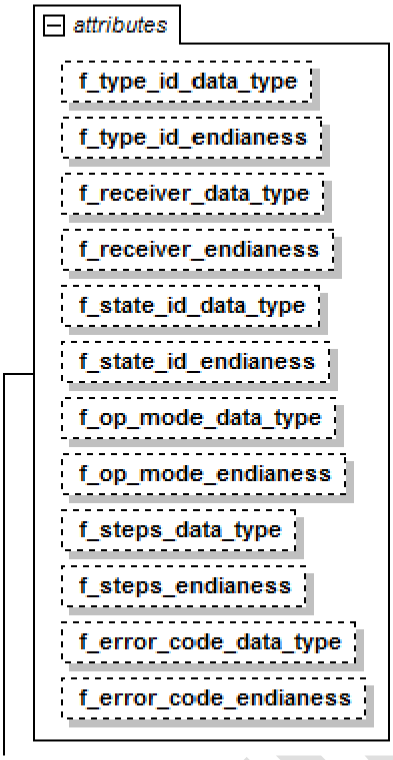


Figure 9: KMatrix element attributes

Attribute name	Description
f_<field_name>_data_type	The data type of the field <field_name> as integer (see 3.1.9 for corresponding data type).
f_<field_name>_endianness	The endianness of the field <field_name>. "little" means little endian, "big" means big endian.

Table 139: KMatrix element attributes

Attribute name	Description
canId	The CAN identifier in the header of the CAN message
senderRef	The DCP id of the sending DCP slave. For state change & information PDUs this is fix, because only the master (DCP id = 0) can send this PDU.
length	The length of the CAN payload field.
f_<field_name>	The starting byte of <field_name> in the CAN payload. "H" means that this field is not contained in the CAN payload and its value can be determined by the CAN identifier in the header.

Table 140: Attributes of every state change, information, notification & response PDU

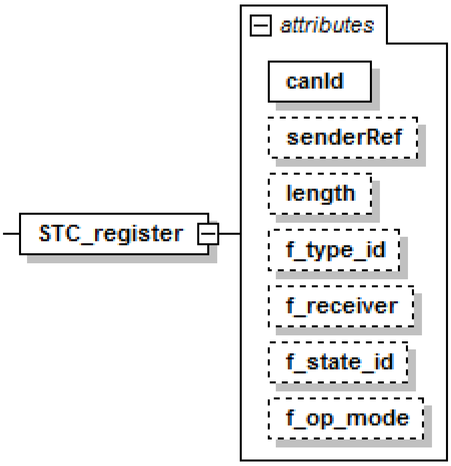


Figure 10: STC_register PDU

The element DAT_input_output consists of up to eight Payload and up to 254 Receiver elements. It is defined as follows:

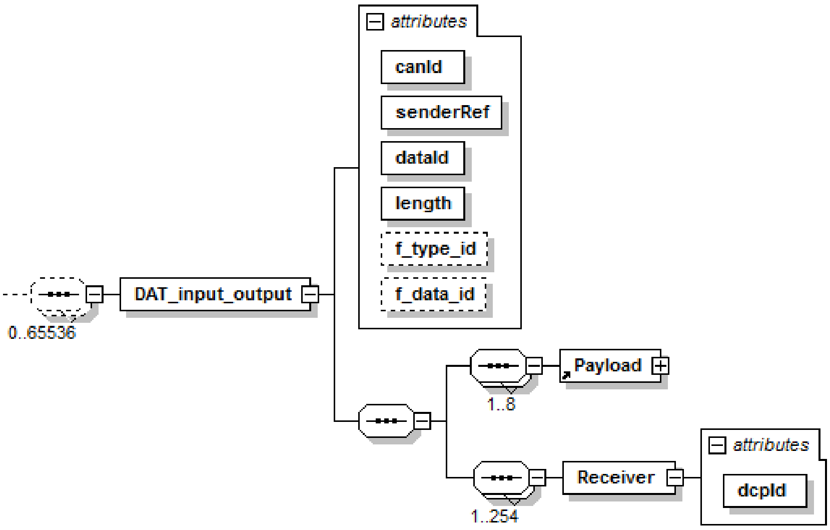


Figure 11: DAT_input_output element

Attribute name	Description
canId	The CAN identifier in the header of the CAN message
senderRef	The DCP id of the sending DCP slave.
dataId	The data id of the DAT_input_output PDU.
length	The length of the CAN payload field.
f_<field_name>	The starting byte of <field_name> in the CAN payload. “H” means that this field is not contained in the CAN payload and its value can be determined by the CAN identifier in the header.
dcpId	The DCP id of the DCP slave which receives this CAN message.

Table 141: DAT_input_output element attributes

In DAT_input_output, the Payload element contains the definition of one output from the sending DCP slave. The choice of the Payload element defines the data type of the output. Payload is defined as follows:

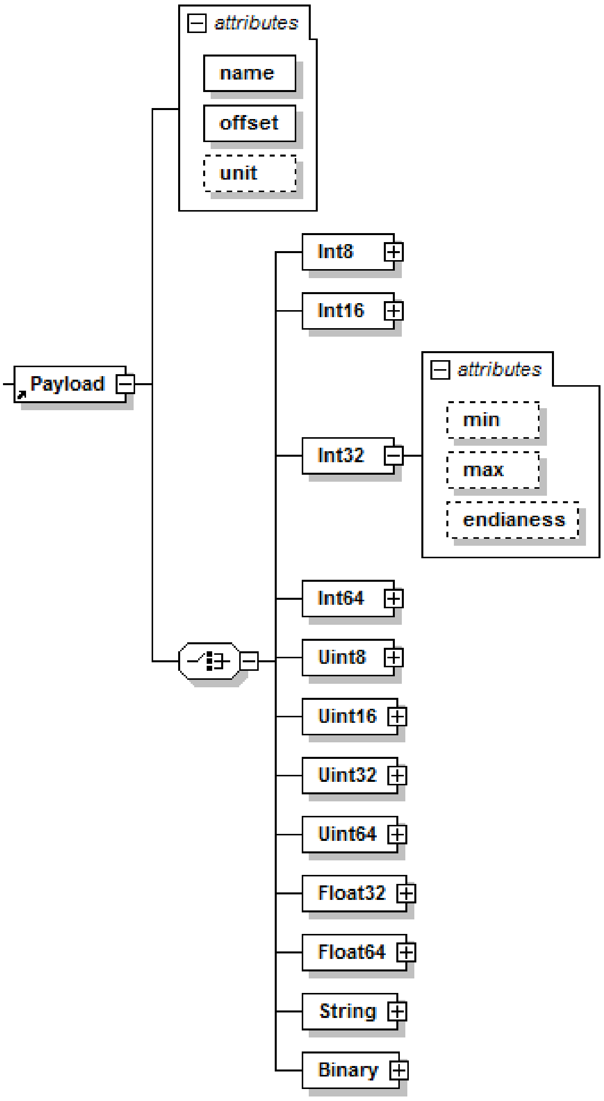


Figure 12: Payload element

Attribute name	Description
name	The name of the output.
offset	The starting byte of the output in the CAN payload. Note: This is not equal to the position in the CFG_output PDU.
unit	The unit of the send output.
min	The minimum of the output
max	The maximum of the output.
endianness	The endianness of the output. “little” means little endian, “big” means big endian.

Figure 13: Payload element attributes

The element DAT_parameter consists of up to eight Payload and up to 254 Receiver elements. It is defined as follows.

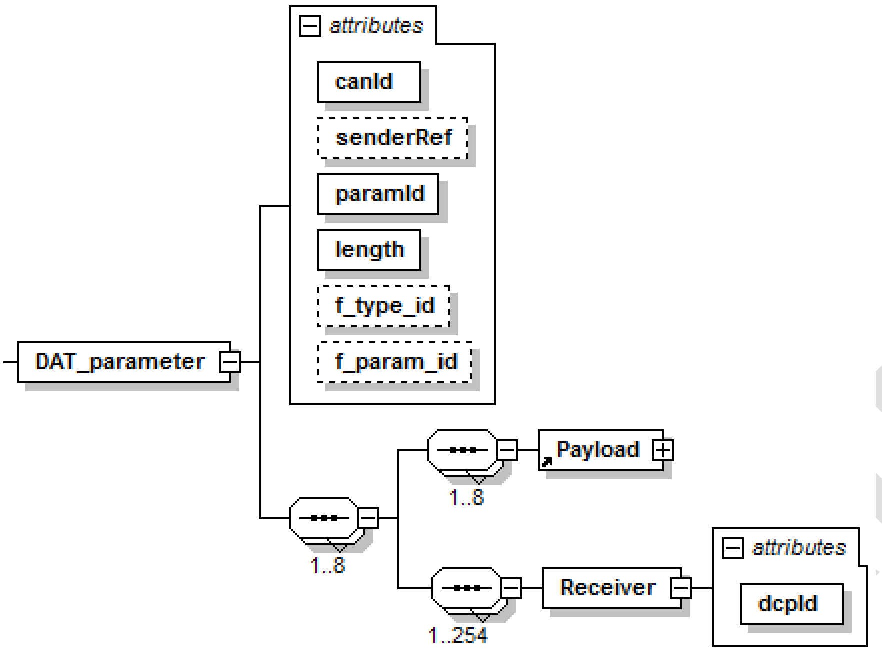


Figure 14: DAT_parameter element

Attribute name	Description
canId	The CAN identifier in the header of the CAN message
senderRef	The DCP id of the sending DCP slave.
paramId	The parameter id of the DAT_parameter PDU.
length	The length of the CAN payload field.
f_<field_name>	The starting byte of <field_name> in the CAN payload. "H" means that this field is not contained in the CAN payload and its value can be determined by the CAN identifier in the header.
dcpId	The DCP id of the DCP slave which receives this CAN message.

Table 142: Attributes of DAT_parameter

4.5.4 Definition of the Scenario Configuration

The element `ScenarioConfiguration` is defined as follows:

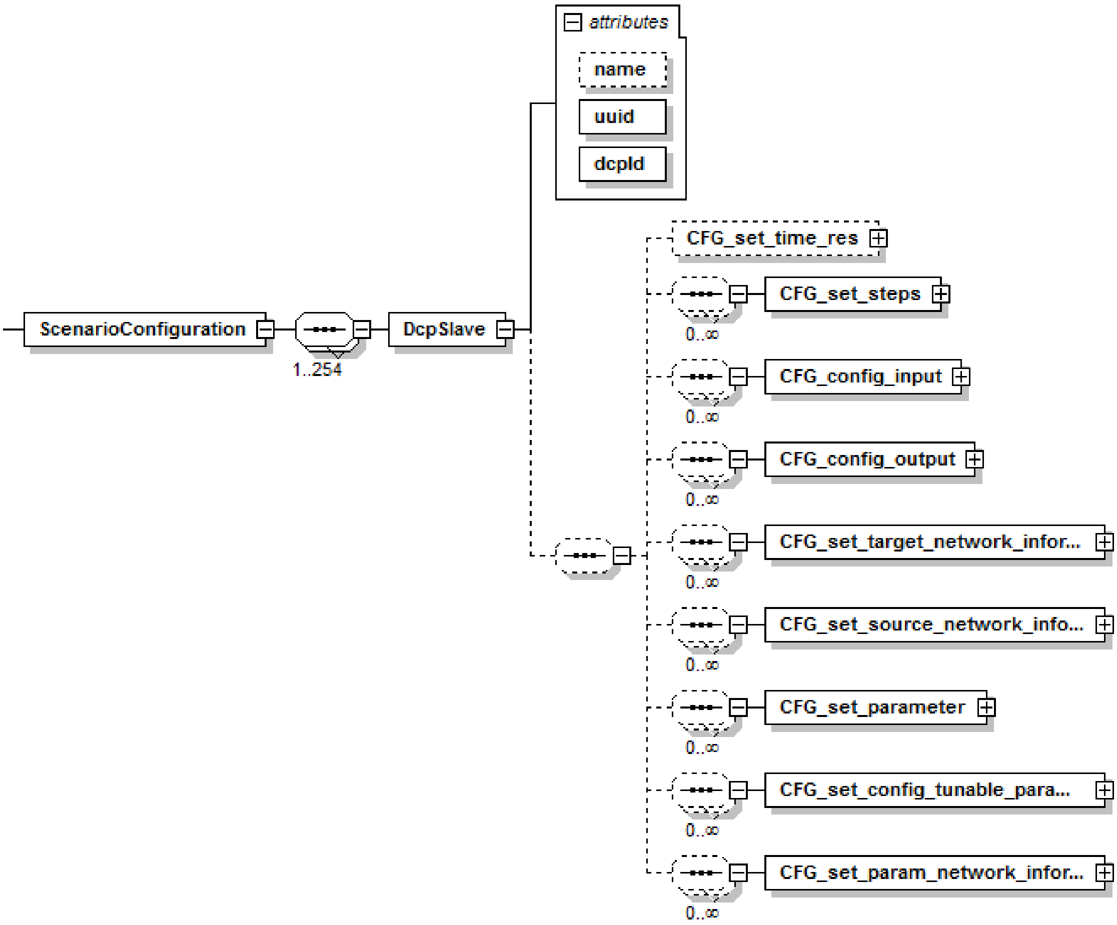


Figure 15: ScenarioConfiguration element

Attribute name	Description
name	The name of the DCP slave.
uuid	The uuid of the DCP slave.
dcpId	The DCP id of the DCP slave.
All other attributes	See section 3.4 for further description.

Table 143: Attributes of ScenarioConfiguration and subsequent elements

5 DCP Slave Description

5.1 General

All static information related to a DCP slave is stored in a text file in XML format. This is called DCP slave description. The provider of a DCP slave shall ensure that the accompanying DCP slave description reflects the implementation of the delivered DCP slave. The DCP slave description shall be consistent with the delivered DCP slave at all times. The medium or distribution mechanism of the DCP slave description is arbitrary.

The file extension of a DCP slave description file is DCPX. The structure of this XML file is defined using the schema file `dcpSlaveDescription.xsd`. This schema file utilizes the following helper schema files.

```
dcpAnnotation.xsd
dcpAttributeGroups.xsd
dcpType.xsd
dcpDataType.xsd
dcpUnit.xsd
dcpVariable.xsd
```

These XSD schema files comply with the XSD 1.1 specification [7]. They contain assertions to support formal verification of dependencies, as stated in this specification document. XSD 1.0 compliant schema files may be created by transformation using an XSLT file (Extensible Stylesheet Language Transformation).

> *Note: This XSLT file is provided together with the XSD schema files to ensure correct transformation. However, it removes the assertions.*

In this section these schema files are discussed. The normative definition are the above mentioned schema files[2]. Below, optional elements are marked with a “dashed” box. The required data types (e.g. xs:normalizedString) are defined in the XML-schema standard (see http://www.w3.org/TR/xmlschema-2 for more information).

5.2 Data Type Definitions

The data types used in the DCP schema files are as follows:

XML	DCP equivalent	Description
xs:unsignedByte	uint8	unsignedByte is derived from unsignedShort by setting the value of maxInclusive to be 255. The base type of unsignedByte is unsignedShort.
xs:unsignedShort	uint16	unsignedShort is derived from unsignedInt by setting the value of maxInclusive to be 65535. The base type of unsignedShort is unsignedInt.
xs:unsignedInt	uint32	unsignedInt is derived from unsignedLong by setting the value of maxInclusive to be 4294967295. The base type of unsignedInt is unsignedLong.
xs:unsignedLong	uint64	unsignedLong is derived from nonNegativeInteger by setting the value of maxInclusive to be 18446744073709551615. The base type of unsignedLong is nonNegativeInteger.
xs:byte	int8	int is derived from long by setting the value of maxInclusive to be 2147483647 and minInclusive to be -2147483648. The base type of int is long.

[2] The screenshots of this section have been generated from the schema files with the tool “Altova XMLSpy”. Please see www.altova.com.

xs:short	int16	short is derived from int by setting the value of maxInclusive to be 32767 and minInclusive to be -32768. The base type of short is int.
xs:int	int32	int is derived from long by setting the value of maxInclusive to be 2147483647 and minInclusive to be -2147483648. The base type of int is long.
xs:long	int64	long is derived from integer by setting the value of maxInclusive to be 9223372036854775807 and minInclusive to be -9223372036854775808. The base type of long is integer.
xs:float	float32	float is patterned after the IEEE single-precision 32-bit floating point type (see IEEE 754-1985)
xs:double	float64	The double datatype is patterned after the IEEE double-precision 64-bit floating point type (see IEEE 754-1985)
xs:normalizedString	string	String without carriage return, line feed, and tab characters.
xs:dataTime	Implementation specific	Date, time and time zone *Example: 2002-10-23T12:00:00Z* *(noon on October 23, 2002, Greenwich Mean Time)*

Table 144: DCP slave description data types

The first line of the DCP slave description must contain its encoding scheme. It is required that the encoding scheme is always UTF-8:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DCP schema files (*.xsd) are also stored in UTF-8.

> *Note: The definition of an encoding scheme is a prerequisite, in order for the XML file to contain letters outside of the 7 bit ANSI ASCII character set, such as German umlauts, or Asian characters.*

The special values NAN, +INF, -INF for variables values are not allowed in the DCP xml files.

> *Note: Child information items, such as elements of sequences are ordered lists according to document order, whereas attribute information items are unordered sets (see http://www.w3.org/TR/XML-infoset/#infoitem.element). The DCP slave description schema is based on ordered lists in a sequence and therefore parsing must preserve this order.*

5.3 Definition of dcpSlaveDescription Element

This is the root level schema file and contains the following definition (the figure below shows all elements in the schema file. Data is defined as attributes to these elements, not shown in this figure).

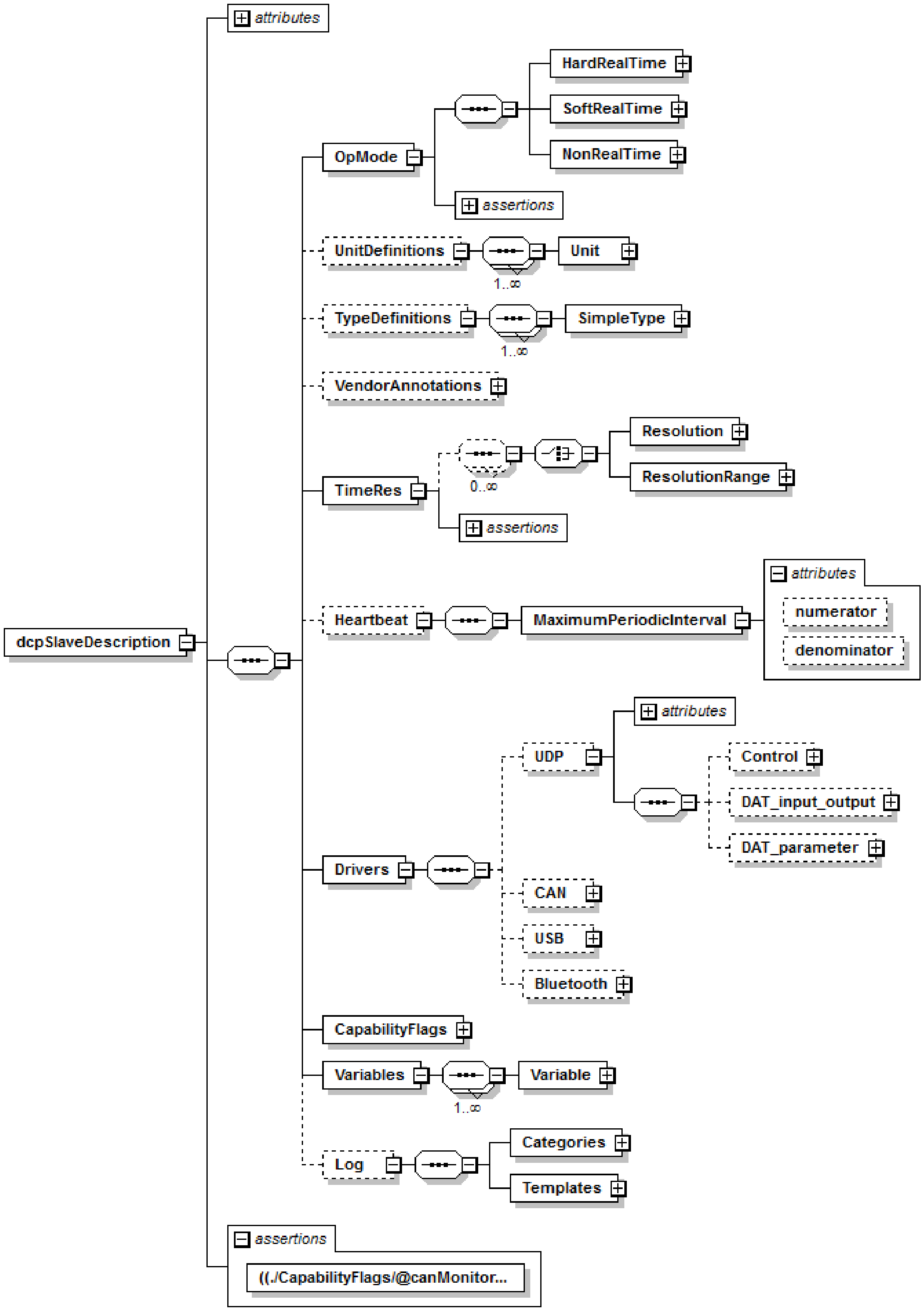


Figure 16: DCP slave description root level XSD schema

On the top level, the schema consists of the following elements.

Element name	Description
OpMode	Valid operating modes of the described DCP slave.
UnitDefinitions	A global list of unit and display unit definitions (see section 5.5).
TypeDefinitions	A global list of type definitions.

VendorAnnotations	Additional vendor specific data. May be ignored.
TimeRes	A list of permissible single time resolutions and resolution ranges.
Heartbeat	If present, the DCP slave uses the given settings to monitor a heartbeat signal provided by the DCP master.
Drivers	This element contains information for all available DCP slave DCP drivers.
CapabilityFlags	The attributes under this element indicate a DCP slave’s capabilities.
Variables	The central DCP slave data structure defining all variables of the DCP slave that are visible/accessible via DCP.
Log	This element contains categories and templates for logging.

Table 145: DCP slave description root level elements

The XML attributes of `dcpSlaveDescription` are as follows.

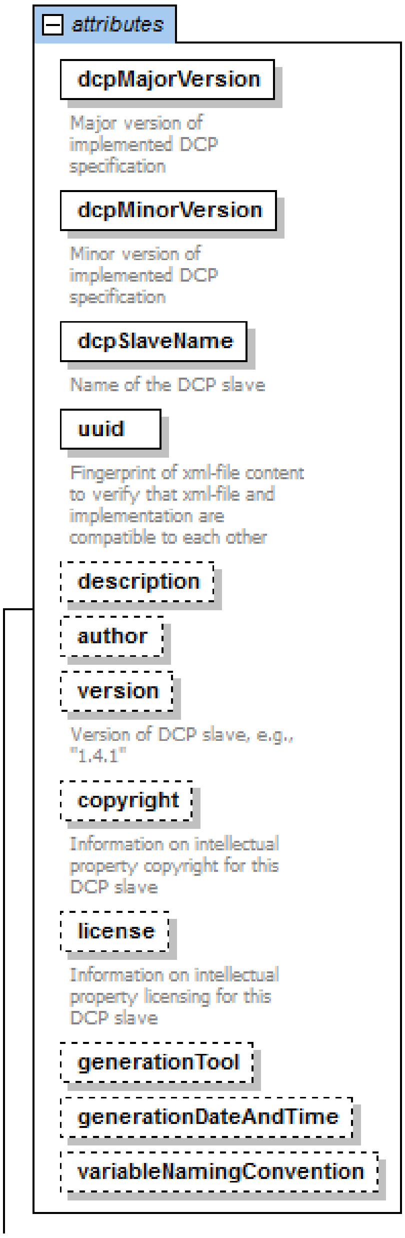

attributes
dcpMajorVersion
Major version of implemented DCP specification
dcpMinorVersion
Minor version of implemented DCP specification
dcpSlaveName
Name of the DCP slave
uuid
Fingerprint of xml-file content to verify that xml-file and implementation are compatible to each other
description
author
version
Version of DCP slave, e.g., "1.4.1"
copyright
Information on intellectual property copyright for this DCP slave
license
Information on intellectual property licensing for this DCP slave
generationTool
generationDateAndTime
variableNamingConvention

Table 146: dcpSlaveDescription element attributes

Attribute name	Description
dcpMajorVersion	The DCP major version that was used to generate the DCPX file and accompanying DCP slave. See section 3.1.2.
dcpMinorVersion	The DCP minor version that was used to generate the DCPX file and accompanying DCP slave. See section 3.1.2.
dcpSlaveName	The name of the complete DCP slave.
uuid	The *universally unique identifier* is a string that is used to uniquely identify a DCP slave in a global environment. The `uuid` acts as a fingerprint of relevant information. Typically, the `uuid` is assigned when the DCP slave description file is generated. It is used for verification during the registration process of a DCP slave.
description	Optional string that contains a brief description of the complete DCP slave.
author	Optional string that contains name and organization of the DCP slave author.
version	Optional development version number of the DCP slave.
copyright	Optional information on the intellectual property copyright for this DCP slave.
license	Optional information on the intellectual property licensing for this DCP slave.
generationTool	Optional information about the tool the DCPX file was generated with.
generationDateAndTime	Optional date and time when the DCPX file was generated. The format is a subset of "xs:dateTime" and should be: "YYYY-MM-DDThh:mm:ssZ" (with one "T" between date and time; "Z" characterizes the Zulu time zone, in other words Greenwich meantime).
variableNamingConvention	Defines whether the variable names in Variables/Variable/name and in TypeDefinitions/SimpleType/name follow a particular convention. Available options are: flat: A string (the default). structured: Names including "." as hierarchy separator. See section 3.1.17.1 for details.

Table 147: dcpSlaveDescription element attributes

5.4 Definition of OpMode Element

Element name	Description
HardRealTime	If its boolean attribute `set` equals true, the DCP slave is capable of operating in hard real time mode.
SoftRealTime	If its boolean attribute `set` equals true, the DCP slave is capable of operating in soft real time mode.
NonRealTime	If its boolean attribute `set` equals true, the DCP slave is capable of operating in non real time mode.

Table 148: Operating modes

At least one of the available operating modes must be set to true.

5.5 Definition of UnitDefinitions Element

> *Note: This definition of units corresponds with the definition of units in FMI 2.0* [8]. *This section recapitulates the most important definitions for completeness. For examples and the full definitions, see the FMI 2.0 specification document.*

The `UnitDefinitions` element is specified as shown in Figure 17. If this element is present, it contains one or more Unit elements.

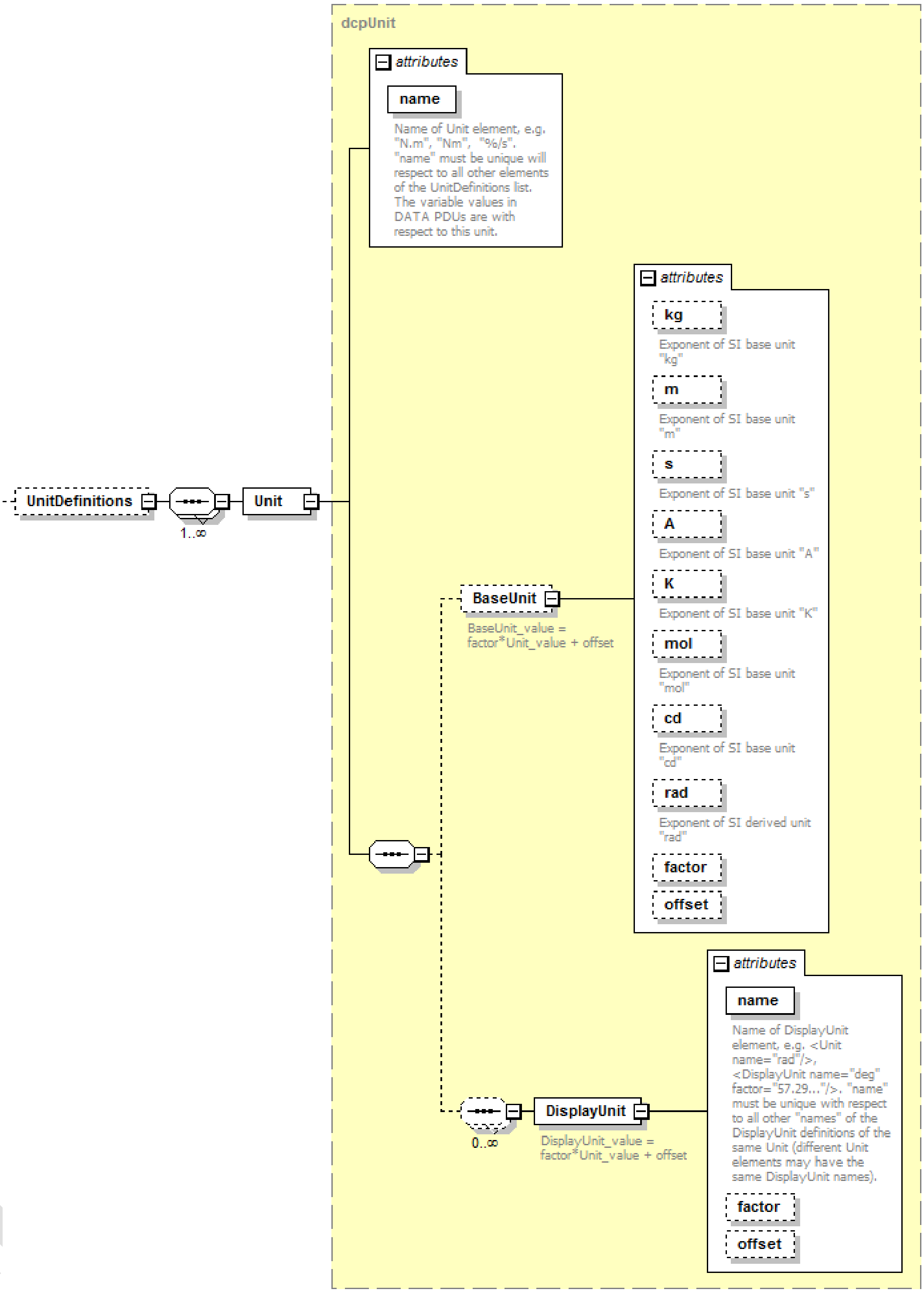

dcpUnit
attributes
name
Name of Unit element, e.g. "N.m", "Nm", "%/s". "name" must be unique will respect to all other elements of the UnitDefinitions list. The variable values in DATA PDUs are with respect to this unit.
UnitDefinitions
1..∞
Unit
attributes
kg
Exponent of SI base unit "kg"
m
Exponent of SI base unit "m"
s
Exponent of SI base unit "s"
A
Exponent of SI base unit "A"
K
Exponent of SI base unit "K"
mol
Exponent of SI base unit "mol"
cd
Exponent of SI base unit "cd"
rad
Exponent of SI derived unit "rad"
factor
offset
BaseUnit
BaseUnit_value = factor*Unit_value + offset
attributes
name
Name of DisplayUnit element, e.g. <Unit name="rad"/>, <DisplayUnit name="deg" factor="57.29..."/>. "name" must be unique with respect to all other "names" of the DisplayUnit definitions of the same Unit (different Unit elements may have the same DisplayUnit names).
factor
offset
0..∞
DisplayUnit
DisplayUnit_value = factor*Unit_value + offset

Figure 17: UnitDefinitions Element

Element name	Description
name	A name of String data type.

Table 149: Unit element attributes

The Unit element contains one BaseUnit element, having the attributes defined in Table 150.

Attribute name	Description

kg	Optional attribute of integer data type. Its default value is zero.
m	Optional attribute of integer data type. Its default value is zero.
s	Optional attribute of integer data type. Its default value is zero.
A	Optional attribute of integer data type. Its default value is zero.
K	Optional attribute of integer data type. Its default value is zero.
mol	Optional attribute of integer data type. Its default value is zero.
cd	Optional attribute of integer data type. Its default value is zero.
rad	Optional attribute of integer data type. Its default value is zero.
factor	Optional attribute of double data type. Its default value is 1.0.
offset	Optional attribute of double data type. Its default value is 0.0.

Table 150: Base unit element attributes

A value with respect to Unit (abbreviated as “Unit_value”) is converted with respect to BaseUnit (abbreviated as “BaseUnit_value”) using the equation:

BaseUnit_value = factor * Unit_value + offset

The `Unit` element contains an optional `DisplayUnit` element. It contains the attributes as defined in Table 151.

Attribute name	Description
name	Attribute of normalizedString data type.
factor	Optional attribute of double data type. Its default value is 1.0.
offset	Optional attribute of double data type. Its default value is 0.0.

Table 151: DisplayUnit element attributes

5.6 Definition of TypeDefinitions Element

5.6.1 General

The TypeDefinitions element is defined as follows.

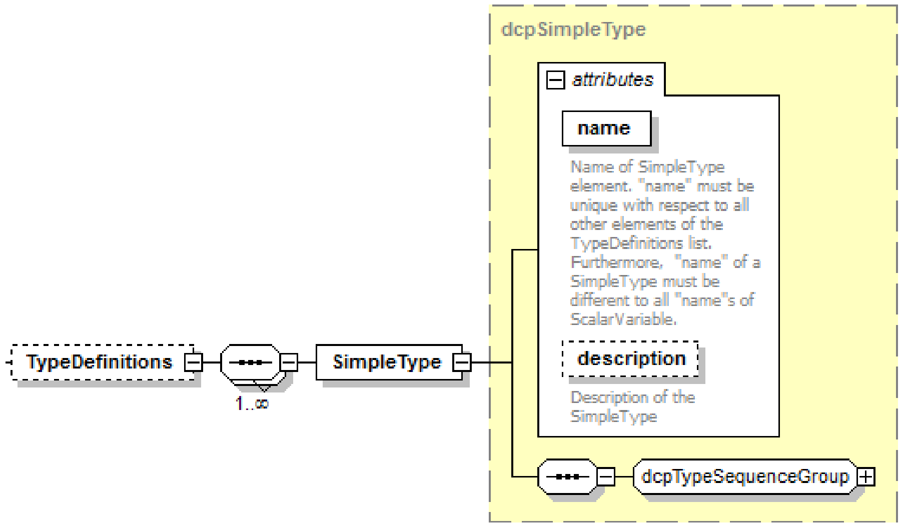

dcpSimpleType
attributes
name
Name of SimpleType element. "name" must be unique with respect to all other elements of the TypeDefinitions list. Furthermore, "name" of a SimpleType must be different to all "name"s of ScalarVariable.
description
Description of the SimpleType
TypeDefinitions
1..∞
SimpleType
dcpTypeSequenceGroup

Figure 18: TypeDefinitions Element

The optional TypeDefinitions element includes a list of SimpleType elements having the list of attributes defined in Table 152.

Attribute name	Description
name	Name of this simple type.
description	An optional description.

Table 152: SimpleType element attributes

5.6.2 Definition of Data Types and Attributes

A sequence group defines the available DCP data types and their attributes. A choice element is provided inside a dcpVariableSequenceGroup, as shown in figure Figure 19.

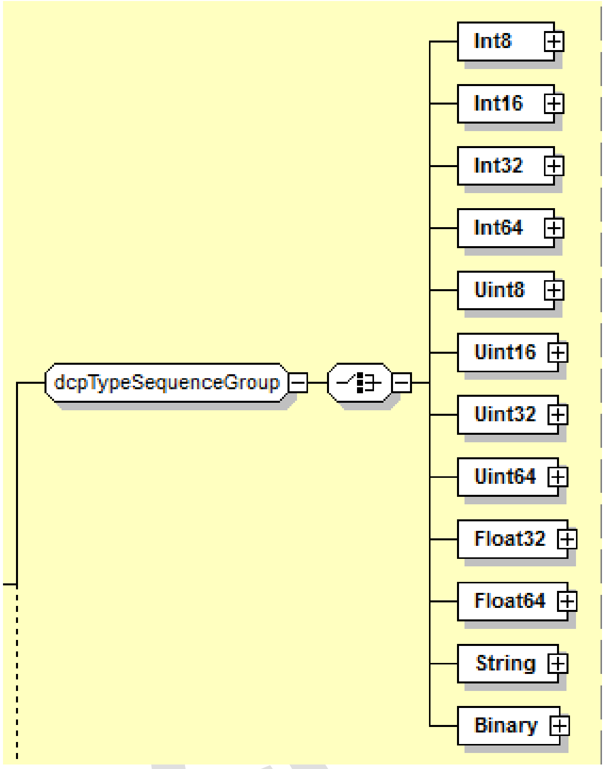


Figure 19: Type sequence group

All unsigned integer (data type id 0 to 3), integer (data type id 4 to 7) float (data type id 8 and 9), string (data type id 10) and binary (data type id 11) data types have subsets of different attributes. For data type definitions see section 3.1.9.

Attribute	Description
declaredType	The declared type of the variable.
displayUnit	Default display unit. The conversion to the "unit" is defined with the element "<dcpSlaveDescription><UnitDefinitions>". If the corresponding "displayUnit" is not defined under "<UnitDefinitions><Unit><DisplayUnit>", then displayUnit is ignored.
gradient	The gradient attribute indicates that the DCP slave variable is designed to change at a maximum permissible rate. Its unit is 1/s, and the data type corresponds to the variable data type. This attribute is optional.
max	The max attribute indicates that the DCP slave variable is designed to operate at or below an upper bound value (Variable value >= max). The data type of this attribute corresponds to the parent element. This attribute is optional.
maxConsecMissedPdus	This defines the maximum number of acceptable consecutively missed PDUs.
maxSize	This defines the maximum size of the data type in bytes. Its data type is unsigned integer.
mimeType	The MIME type of the data type.

min	The min attribute indicates that the DCP slave variable is designed to operate at or above a lower bound value (Variable value >= min). The data type of this attribute corresponds to the parent element. This attribute is optional.
nominal	Nominal value of variable. If not defined and no other information about the nominal value is available, then nominal = 1 is assumed. Note: The nominal value of a variable can be, for example used to determine the absolute tolerance for this variable as needed by numerical algorithms: absoluteTolerance = nominal * tolerance *0.01 where tolerance is, e.g., the relative tolerance].
quantity	Physical quantity of the variable, for example “Angle”, or “Energy”. The quantity names are not standardized.
start	Definition of start value. The data type of this attribute corresponds to the parent element.
unit	Unit of the variable defined with UnitDefinitions.Unit.name that is used for the model equations. For example “N.m”: in this case a Unit.name = "N.m" must be present under UnitDefinitions.

Table 153: Attributes of all defined variables.

5.7 Definition of VendorAnnotations Element

The element VendorAnnotations of DCPDescription is defined as follows.

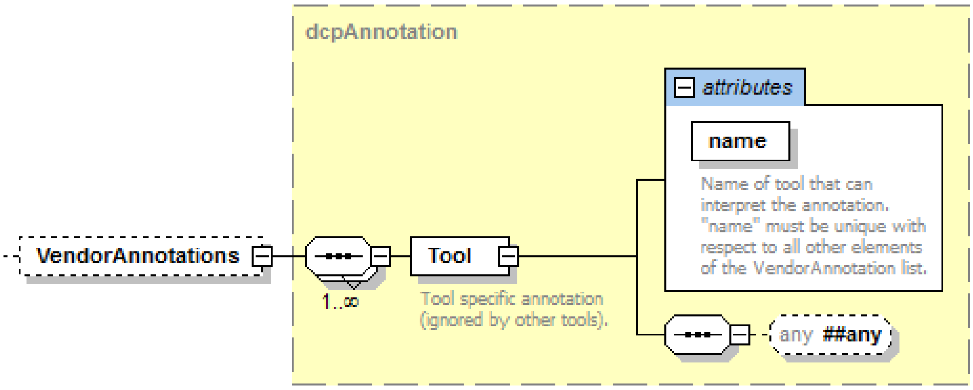

dcpAnnotation
attributes
name
Name of tool that can interpret the annotation. "name" must be unique with respect to all other elements of the VendorAnnotation list.
VendorAnnotations
1..∞
Tool
Tool specific annotation (ignored by other tools).
any ##any

Figure 20: Vendor annotations

VendorAnnotations consists of an ordered set of annotations that are identified by the name of the tool that can interpret the any-element. The any-element may be an arbitrary XML data structure. Attribute name must be unique with respect to all other elements of the VendorAnnotations list.

5.8 Definition of TimeRes Element

The element TimeRes is specified as follows.

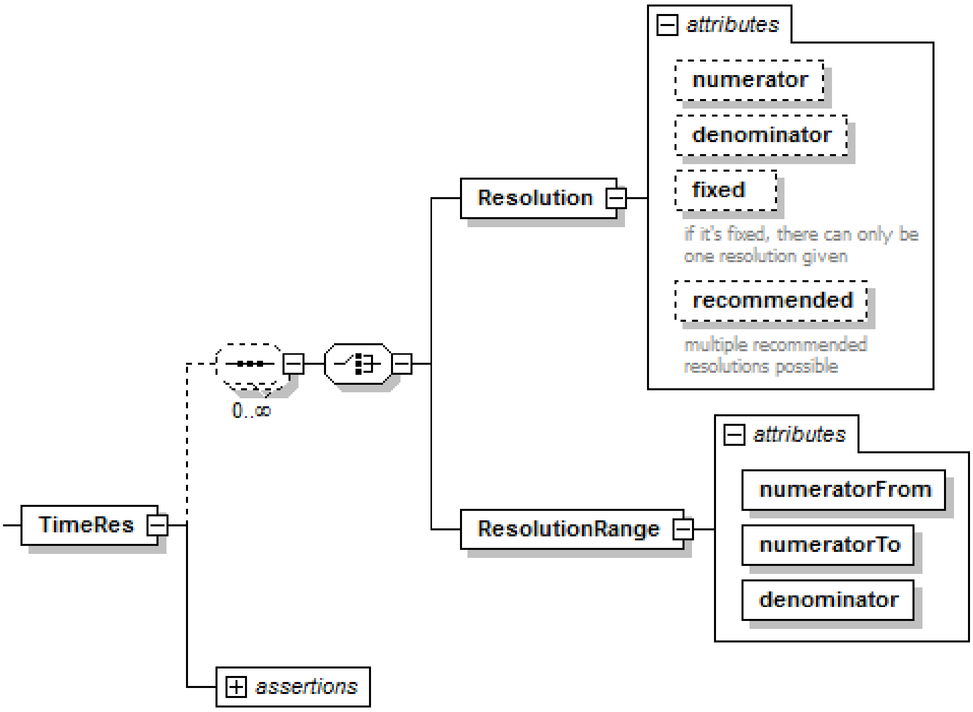


Figure 21: Time resolution element

`TimeRes` contains an ordered sequence of `Resolution` or `ResolutionRange` elements. Their attributes are described as follows.

Attribute name	Description
numerator	Optional numerator value specified as unsigned integer data type. Its default value is 1.
denominator	Optional denominator value specified as unsigned integer data type. Its default value is 1000.
fixed	Optional attribute of boolean data type. Its default value is true. If the fixed value is true, then there can only be one single resolution specified.
recommended	Optional attribute of boolean data type. If the recommended value is true, then this single resolution value is recommended for simulation. Multiple recommended single resolution values are possible.
numeratorFrom	This attribute specifies the begin of a numerator resolution range. Its data type is unsigned integer.
numeratorTo	This attribute specifies the end of a numerator resolution range. Its data type is unsigned integer.
denominator	This attribute specifies the denominator for one resolution range. Its data type is unsigned integer.

Table 154: Time resolution attributes

5.9 Definition of Heartbeat Element

The optional `Heartbeat` element is specified as follows. It contains a single `MaximumPeriodicInterval` element with 2 attributes.

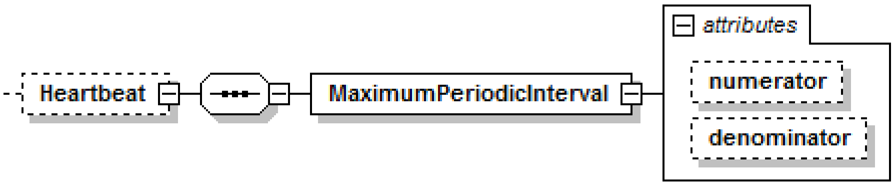


Figure 22: Heartbeat element

Attribute name	Description
numerator	Optional attribute of unsigned integer data type. Its default value is 1.
denominator	Optional attribute of unsigned integer data type. Its default value is 1.

Table 155: Heartbeat element attributes

5.10 Definition of Drivers

Each transport protocol listed under the `Drivers` element defines a maximumTransmissionUnit attribute.

Attribute name	Description
maximumTransmissionUnit	Optional attribute of unsigned integer data type.

5.10.1 UDP/IPv4

The elements and attributes for UDP via IPv4 are shown in Figure 23 and are defined as follows.

The UDP via IPv4 transport protocol allows separate configurations for different PDU families.

- The optional Control element defines the host and port attributes intended for receiving PDUs of the Control PDU family.
- The optional DAT_input_output element defines a host attribute, as well as an optional list of either port ranges, indicated by from and to attributes, or single ports, indicated by the port attribute. This is intended for receiving DAT_input_output PDUs.
- The optional DAT_parameter element defines a host attribute, as well as an optional list of either port ranges, indicated by from and to attributes, or single ports, indicated by the port attribute. This is intended for receiving DAT_parameter PDUs.

These recurring attributes are specified in Table 156.

Attribute name	Description
host	Optional attribute of normalizedString data type. This attribute may contain one of the following: • An IP according to RFC 791 [9] • A hostname according to RFC 952 [10] • A fully qualified domain name according to RFC 1035 [11]
port	Attribute of unsigned short data type.
from	Attribute of unsigned short data type.
to	Attribute of unsigned short data type.

Table 156: Control element attributes

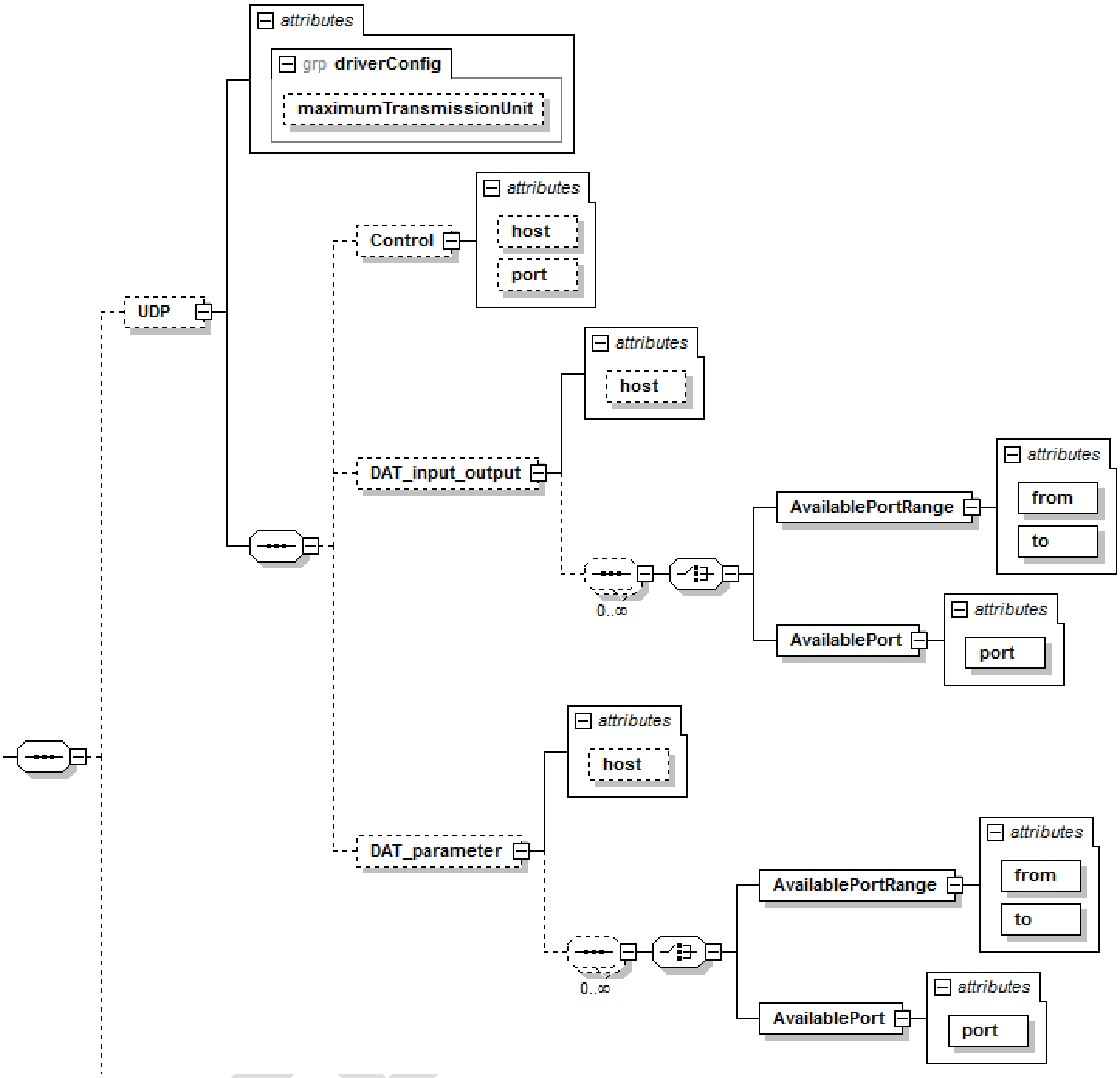


Figure 23: UDP/IPv4 elements and attributes

5.10.2 CAN

The DCP specification for CAN bus is defined in a non-native way. No entries are needed here.

5.10.3 USB

The elements and attributes for USB are defined in Table 157.

Attribute name	Description
maxPower	Optional attribute of unsignedByte data type.

Table 157: USB element attributes

The USB element contains an optional list of DataPipe elements, having the attributes defined in Table 158.

Attribute name	Description
direction	Optional attribute of string data type, enumerated with either “In” or “Out”.

endpointAddress	Attribute of unsignedByte data type. Its value must be greater than 2.
interval	Attribute of unsignedByte data type.

Table 158: DataPipe element attributes

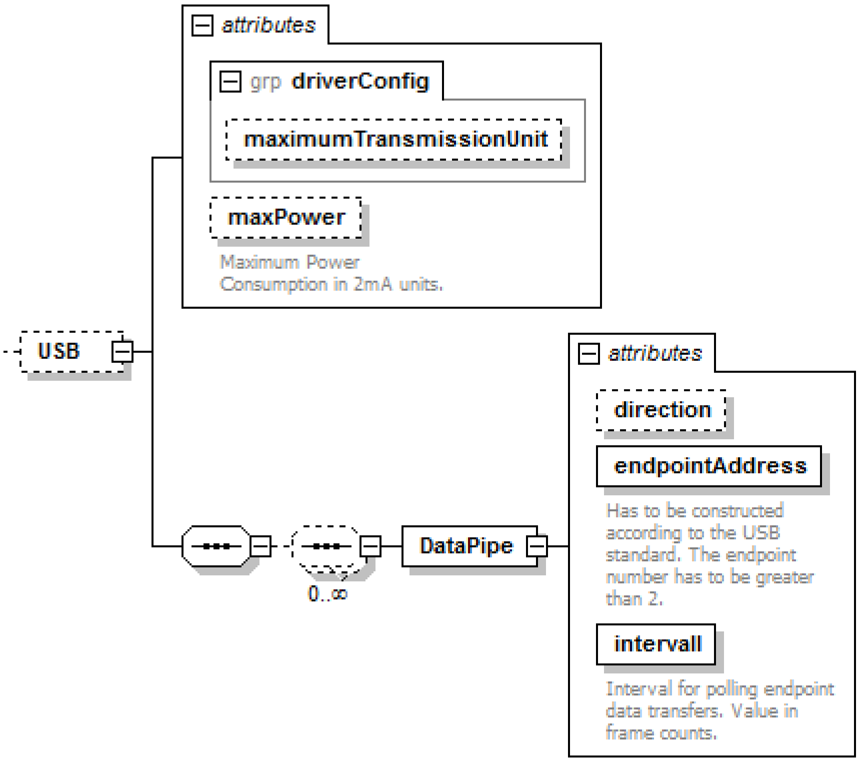

attributes
grp driverConfig
maximumTransmissionUnit
maxPower
Maximum Power Consumption in 2mA units.
USB
0..∞
DataPipe
attributes
direction
endpointAddress
Has to be constructed according to the USB standard. The endpoint number has to be greater than 2.
intervall
Interval for polling endpoint data transfers. Value in frame counts.

Figure 24: USB element

5.10.4 Bluetooth

The elements and attributes for `Bluetooth` are shown in Figure 25 and defined as follows.

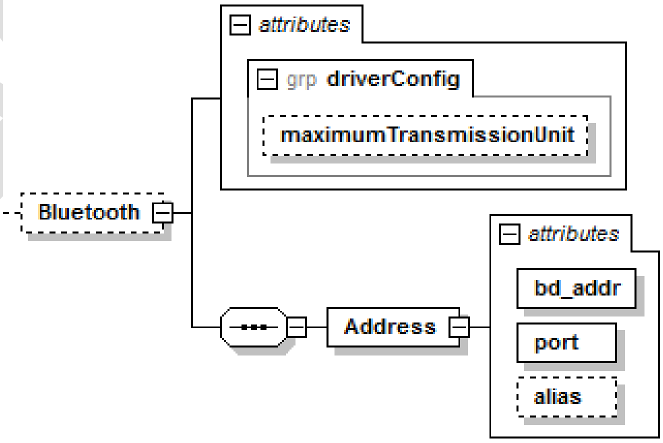


Figure 25: Bluetooth element

The attributes of the `Address` element are specified in Table 159.

Attribute name	Description
bd_addr	Attribute of String data type. Its value must comply with the following regular expression:

	([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})
port	Attribute of unsignedByte data type. Its value is specified to be larger or equal to 1, and smaller or equal to 30.
alias	Optional attribute of type normalizedString.

Table 159: Address element attributes

5.11 Definition of CapabilityFlags Element

The element CapabilityFlags is defined as follows.

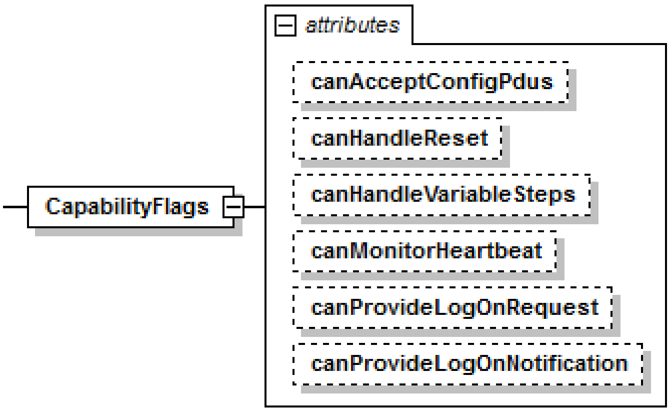


Figure 26: Capability flag element and attributes

The following capability flags are defined to indicate the availability of specific functionality. Each capability flag is of data type bool.

Attribute name	Description
canAcceptConfigPdus	Indicates that a DCP slave is able to process CFG_input, CFG_output, CFG_target_network_information, and CFG_source_network_information PDUs properly. Otherwise, (1) the DCP slaves addressing needs to be resolved in a different way, e.g. manually, and (2) DAT_input_output PDUs need to be configured in a different way, e.g. manually.
canHandleReset	The DCP slave can handle a STC_reset PDU and reset the state machine from STOPPED to CONFIGURATION. Otherwise it waits for STC_deregister to transition to ALIVE.
canHandleVariableSteps	Indicates that the DCP slave can handle variable steps in NRT operating mode.
canMonitorHeartbeat	Indicates that a DCP slave is able to monitor a periodic heartbeat signal sent by a DCP master.
canProvideLogOnNotification	Indicates that the DCP slave supports logging using notifications.
canProvideLogOnRequest	Indicates that the DCP slave supports logging using the request-response mechanism.

Table 160: Capability flags

5.12 Definition of Variables Element

The Variables element contains the following attributes and elements.

5.12.1 Definition of Variable Element

The Variables element contains a sequence of four different elements. Figure 27 shows the structure within the DCP slave description.

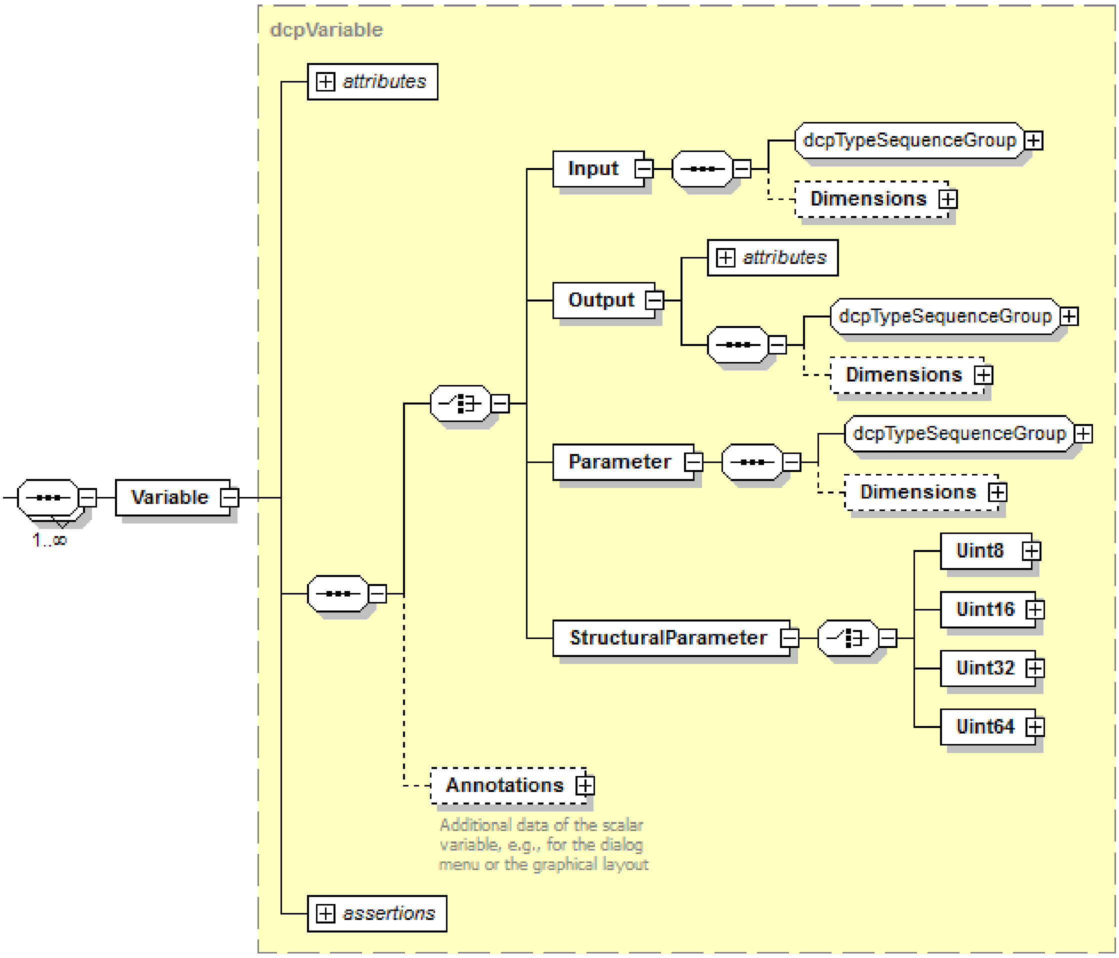


Figure 27: Variable kinds

Element	Description
Input	Inputs of the DCP slave.
Output	Outputs of the DCP slave.
Parameter	Parameters of the DCP slave.
Structural parameter	Structural parameters of the DCP slave.
Annotations	

Table 161: Variable elements

5.12.2 Definition of Variable Element Attributes

The `Variables` element provides statics information on all exposed variables and is defined as follows.

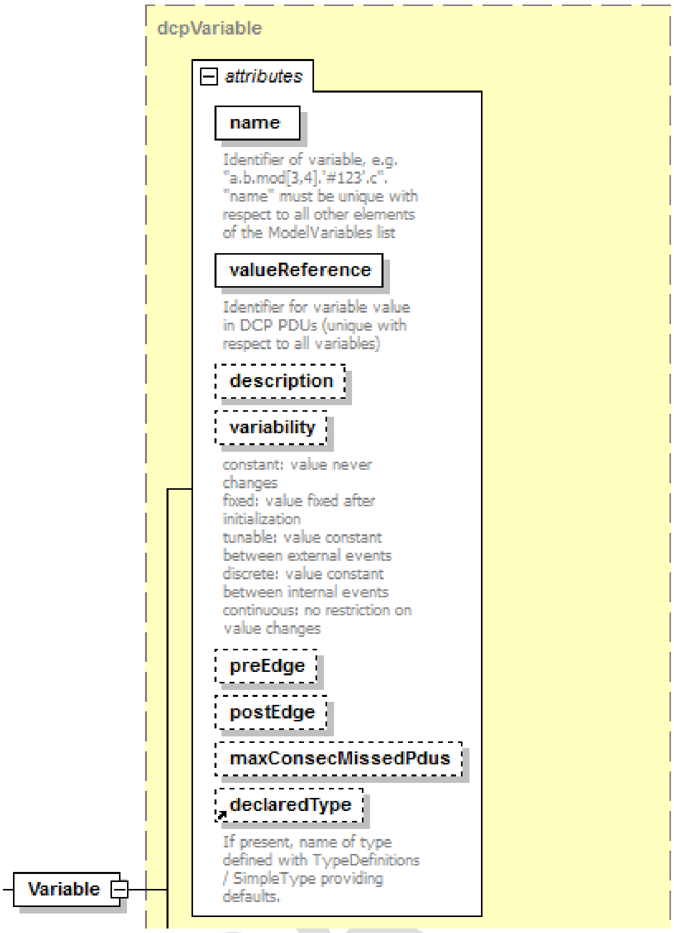

dcpVariable
attributes
name
Identifier of variable, e.g. "a.b.mod[3,4].'#123'.c". "name" must be unique with respect to all other elements of the ModelVariables list
valueReference
Identifier for variable value in DCP PDUs (unique with respect to all variables)
description
variability
constant: value never changes
fixed: value fixed after initialization
tunable: value constant between external events
discrete: value constant between internal events
continuous: no restriction on value changes
preEdge
postEdge
maxConsecMissedPdus
declaredType
If present, name of type defined with TypeDefinitions / SimpleType providing defaults.
Variable

Figure 28: Variable element attributes

Attribute	Description
name	The full name of the variable.
valueReference	A unique handle per DCP slave of the variable, to efficiently identify the variable value in the DCP. This attribute is required.
description	An optional string describing the variable.
variability	Enumeration that defines the time dependency of the variable. It determines the time instants a variable is allowed to change its value. Allowed values: fixed: This setting applies to parameters only. The value of the variable may only be set in state `CONFIGURATION` (see Table 56, `CFG_parameter`). tunable: This setting applies to parameters only. The value of the variable may be set during simulation. See Table 56, `DAT_parameter` PDU. discrete: This setting applies to Inputs and Outputs only. The value of the variable must at least be communicated on a change, but at periodic communication intervals. continuous: This setting applies to Inputs and Outputs only. The value of the variable must be communicated at periodic communication intervals.
preEdge	The preEdge attribute defines the negative maximum deviation from the expected PDU arrival time. Its data type is float64 and its value is given in seconds.

	This attribute is optional.
postEdge	The postEdge attribute defines the positive maximum deviation from the expected PDU arrival time. Its data type is float64 and its value is given in seconds. This attribute is optional.
maxConsecMissedPdus	This attribute defines the maximum allowed number of consecutively missed steps for the given variable. Its data type is uint32.
declaredType	Identifies the name of type defined with TypeDefinitions/SimpleType providing defaults.

Table 162: Variable element attributes

5.12.3 Definition of Variable Data Types and Attributes

Inputs, Outputs and Parameters must be assigned a data type. This is indicated by a corresponding sub-element within a dcpTypeSequenceGroup, as defined in Section 5.6.2.

5.12.4 Definition of Output Element Attributes

Attribute	Description
defaultSteps	This optional attribute specifies the default number of steps of an output. Its data type is unsigned integer. Its default value is 1. This applies to output variables only.
fixedSteps	This optional boolean data type attribute indicates that the number of steps is fixed and cannot be modified. Its default value is true. This applies to output variables only.
minSteps	This optional unsigned integer data type attribute specifies the minimum number of steps. This applies to output variables only.
maxSteps	This optional unsigned integer data type attribute specifies the maximum number of steps. This applies to output variables only.

Table 163: Exclusive output variable attributes

5.12.5 Definition of Multi-Dimensional Data Types

In order to define vector and array data types, the `dimensions` element is a child element of `DCPVariable`. It is defined using a sequence of `dimension` elements, which must have one out of two attributes, either `constant` or `linkedVR`. This is shown in Figure 29.

The assertion is defined as follows:

```
((@constant and not(@linkedVR)) or (not(@constant) and @linkedVR))
```

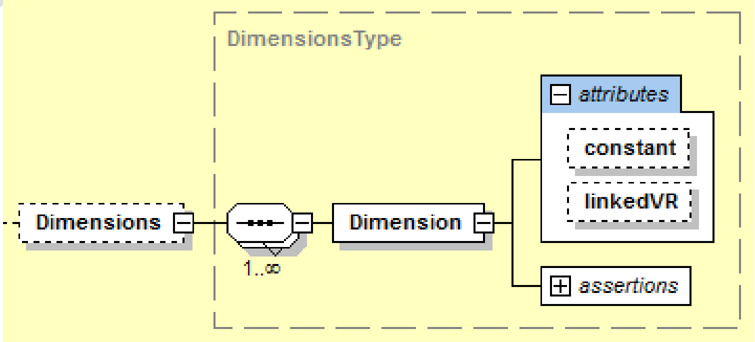


Figure 29: Dimensions element

Table 164 explains the attributes in detail.

Attribute	Description
constant	This is an unsigned long data type attribute, indicating the constant dimension size of a variable.
linkedVR	This is an unsigned long data type attribute, referring to a variable identified through its value reference.

Table 164: Dimension attributes

Figure 30 shows the structural parameter element. It defines four unsigned integer data types (data type id 0 to 3). Its attributes are given in Table 165.

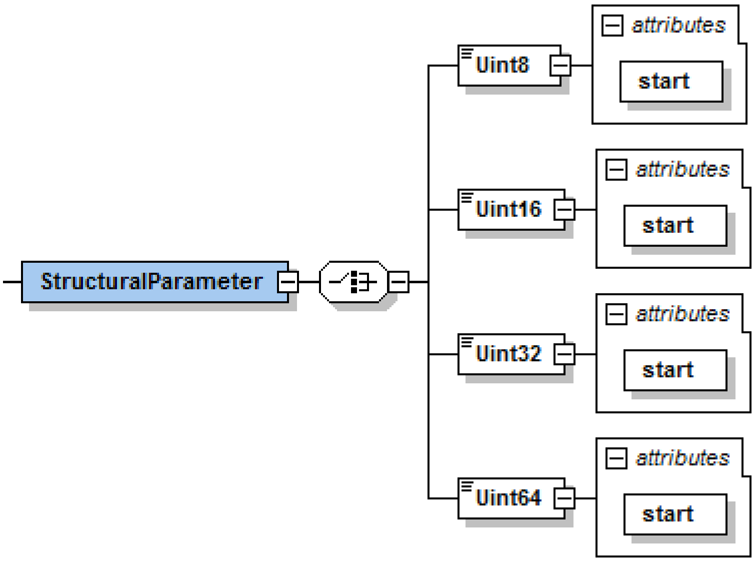


Figure 30: Structural parameter element.

Attribute	Description
start	The start value of the structural parameter variable.

Table 165: Structural parameter element attributes

5.13 Definition of Log Element

For a detailed description of the log mechanisms see section 3.1.22.

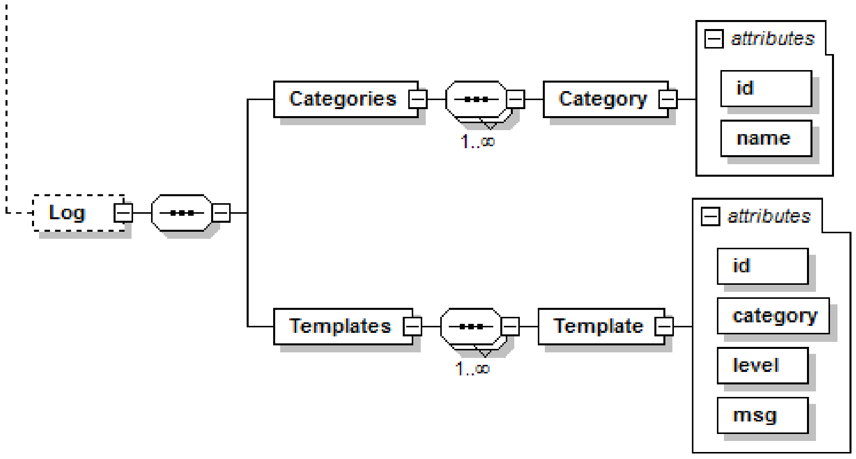


Figure 31: Log Element

Attribute	Description
id	This defines a log category identifier. Its data type is uint8.
name	This defines a name for the log category.

Table 166: Category element attributes

Attribute	Description
id	This defines a log template identifier. Its data type is uint8.
category	This defines a reference to the log category identifier. Its data type is uint8.
level	This defines the log level. Its data type is uint8.
msg	This defines the log message. Its data type is a string, where placeholders starting with "%" followed by the corresponding data type descriptor are used. It is possible to escape "%" by using "%%". *Example: Log message "Initialization at %float32 %%." will result in "Initialization at 35.5 %."*

Table 167: Template element attributes

6 Abbreviations

CAN – Controller area network
CFG – Prefix for Configuration PDUs
DAT – Prefix for Data PDUs
DCP – Distributed co-simulation protocol
DCPX – Distributed co-simulation protocol XML
FMI – Functional mock-up interface
FPGA – Field programmable gate array
HRT – Hard real-time
IEEE - Institute of electrical and electronics engineers
INF – Prefix for Information request PDUs
LoN – Log on notification
LoR – Log on request
LSB – Least significant byte
MSB – Most significant byte
MTU – Maximum transmission unit
NRT – Non real-time
NTF – Prefix for Notification PDUs
PDU – Protocol data unit
RSP – Prefix for Response PDUs
SRT – Soft real-time
STC – Prefix for state change PDUs
UDP – User datagram protocol
USB – Universal serial bus
UTF-8 – Unicode transformation format

7 Literature

[1] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.
[2] Unicode Consortium, *The Unicode Standard, Version 2.0*, no. June. 1996.
[3] “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems,” *IEEE Std 1588-2008*, pp. 1–269, Jul. 2008.
[4] R. Pattis, “EBNF : A Notation to Describe Syntax,” pp. 1–19, 2013.
[5] P. J. Leach, R. Salz, and M. H. Mealling, “A Universally Unique IDentifier (UUID) URN Namespace,” no. 4122. RFC Editor, 2005.
[6] “ISO/DIS 26262:2016 - Road Vehicles - Functional Safety.” International Organization for Standardization, 2016.
[7] “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.” W3C., 2012.
[8] “Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0.” Modelisar Consortium and Modelica Association Project “FMI,” 2014.
[9] J. Postel, “Internet Protocol,” RFC Editor, Sep. 1981.
[10] K. Harrenstien, M. K. Stahl, and E. J. Feinler, “DoD Internet host table specification,” RFC Editor, 1985.
[11] P. Mockapetris, “Domain names - implementation and specification,” RFC Editor, Nov. 1987.

8 Glossary

Term	Description
DCP master	A DCP master is defined as a black box, communicating by DCP PDUs over a given transport protocol, controlling at least one DCP slave.
DCP slave	A DCP slave is defined according to this specification, communicating by DCP PDUs over a given transport protocol, being controlled by exactly one DCP master.
Distributed Co-Simulation Protocol (DCP)	Communication protocol which enables distributed co-simulation.
Distributed Co-Simulation Protocol Description	Description of a DCP slave in XML file format with the file extension DCPX according to specification.
Distributed Co-simulation Scenario (DCS)	A DCS is defined as the integration of multiple DCP slaves to perform a common simulation task.
Error	A DCP slave error is a discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition, that concerns violations of real time requirements or other DCP related functionality.
Failure	Termination of the ability of a DCP slave to perform as intended.
Hard Real-Time	Time-related criteria (deadlines) must be met at all times. Miss of a deadline is assumed to result in a total system failure.
Non-real time simulation	A simulation in which the simulation time is not synchronous to the absolute time.
Protocol Data Unit (PDU)	Information that is exchanged as a unit among DCP slaves.
Real-time simulation	A simulation in which the simulation time is synchronous to the absolute time.

9 Acknowledgment

This project is co-funded by the Austrian Research Promotion Agency (FFG)

This project is co-funded by the Federal Ministry of Education and Research (BMBF)

PROJECT PARTICIPANTS:

Kompetenzzentrum - Das virtuelle Fahrzeug, Forschungsgesellschaft mbH („VIRTUAL VEHICLE") (AT)
AVL List Gmbh (AT)
Spath MicroElectronicDesign (AT)

Dr. Ing. h.c. F. Porsche AG (DE)
Volkswagen AG (DE)
Robert Bosch GmbH (DE)
ETAS GmbH (DE)
dSPACE GmbH (DE)
ESI ITI GmbH (DE)
TWT GmbH Science & Innovation (DE)
RWTH Aachen University (DE)
Technische Universität Ilmenau (DE)
Leibniz University of Hannover (DE)
ks.MicroNova GmbH (DE)

Renault SAS (FR)
Siemens Industry Software SAS (FR)

DISCLAIMER

This ITEA3 Project has been made possible by a financial contribution by the Austrian Research Promotion Agency (FFG) and by the Federal Ministry of Education and Research. The Publication as provided reflects only the authors' view.

Every effort has been made to ensure complete and accurate information concerning this document. However, the author(s) and members of the consortium cannot be held legally responsible for any mistake in printing or faulty instructions. The authors and consortium members retrieve the right not to be responsible for the topicality, correctness, completeness or quality of the information provided. Liability claims regarding damage caused by the use of any information provided, including any kind of information that is incomplete or incorrect, will therefore be rejected. The information contained in this document is based on author's experience and on information received from the project partners.

10 Appendix

A. Key Words to Indicate Requirement Levels

The following definitions are taken from RFC 2119.

MUST: This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

MUST NOT: This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

SHOULD: This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

SHOULD NOT: This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY: This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

Guidance in the use of these Imperatives: Imperatives of the type defined in this memo must be used with care and sparingly. In particular, they MUST only be used where it is actually required for interoperation or to limit behavior which has potential for causing harm (e.g., limiting retransmissions). For example, they must not be used to try to impose a particular method on implementors where the method is not required for interoperability.

Security Considerations: These terms are frequently used to specify behavior with security implications. The effects on security of not implementing a MUST or SHOULD, or doing something the specification says MUST NOT or SHOULD NOT be done may be very subtle. Document authors should take the time to elaborate the security implications of not following recommendations or requirements as most implementors will not have had the benefit of the experience and discussion that produced the specification.

B. Default DCP Slave Integration

The following Figure 32 sketches the procedure for default DCP slave integration using the native DCP specification. A DCP slave provider ships a DCP slave description in DCPX file format and a DCP slave to the DCP integrator. The DCP integrator uses the DCP slave description for configuration of a scenario. This scenario description is exported to a DCP master. The DCP master is an implementation being able to control DCP slaves. It generates a configuration for simulation and rolls out this configuration to the running instances of the DCP slaves. Then the scenario may be simulated.

> *Note: This specification covers the intended behavior of a DCP slave and the DCP slave description. This specification does not cover the import process of DCP slave descriptions, the generation of a valid scenario configuration, the scenario description being exported to the DCP master, the exact steps necessary for DCP slave instantiation, as well as DCP slave implementation details.*

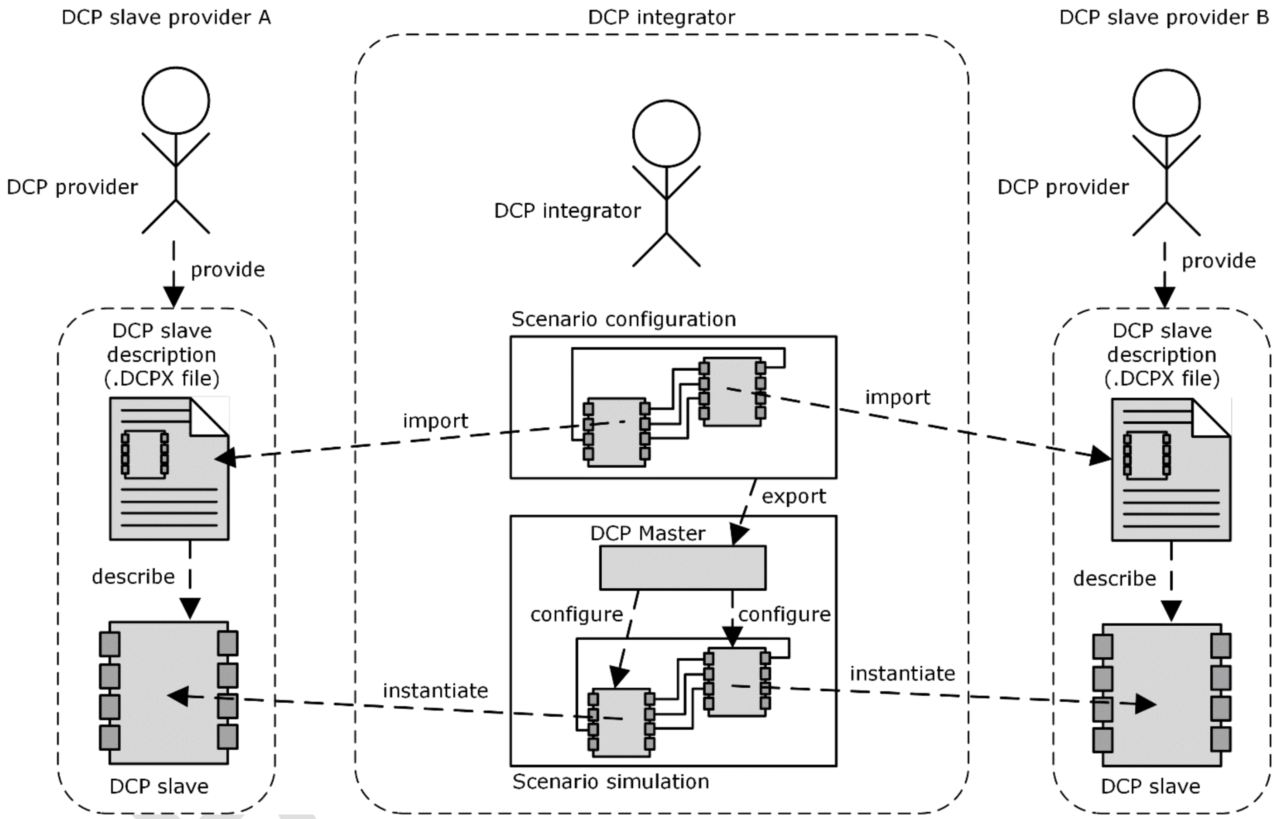


Figure 32: Default DCP slave integration

C. Example: Encoding of Variables

The following tables show the intended encoding of variables when transmitted using the DCP.

Parts of		Coloring
Floating Point	Integer	
Sign	Sign	🟦
Exponent	-	🟩
Fraction	-	🟥

Table 168: Color Indicators

Decimal	42							
Binary	0	0	1	0	1	0	1	0
Hex	0x2A							
	MSB							LSB

Position	n							
DAT_input_output$_{\text{Bin}}$	0	0	1	0	1	0	1	0
DAT_input_output$_{\text{Hex}}$	0x2A							

Table 169: Example uint8

Decimal	7963															
Binary	0	0	0	1	1	1	1	1	0	0	0	1	1	0	1	1
Hex	0x1F								0x1B							
	MSB															LSB

Position	n								n + 1							
DAT_input_output$_{\text{Bin}}$	0	0	0	1	1	0	1	1	0	0	0	1	1	1	1	1
DAT_input_output$_{\text{Hex}}$	0x1B								0x1F							

Table 170: Example uint16

Decimal	335960			
Binary	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 1 0 0 0 0 0	0 1 0 1 1 0 0 0
Hex	0x00	0x05	0x20	0x58
	MSB			LSB

Position	n	n + 1	n + 2	n + 3
DAT_input_output$_{\mathrm{Bin}}$	0 1 0 1 1 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 0 0 0
DAT_input_output$_{\mathrm{Hex}}$	0x58	0x20	0x05	0x00

Table 171: Example uint32

Decimal	622553314543962266							
Binary	0 0 0 0 1 0 0 0	1 0 1 0 0 0 1 1	1 1 0 0 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 0 1 1 1 1	0 0 1 0 0 0 1 0	1 0 0 0 1 0 0 0	1 0 0 1 1 0 1 0
Hex	0x08	0xA3	0xC0	0xF0	0xEF	0x22	0x88	0x9A
	MSB							LSB

Position	n	n + 1	n+ 2	n + 3	n + 4	n + 5	n + 6	n + 7
DAT_input_output$_{\mathrm{Bin}}$	1 0 0 1 1 0 1 0	1 0 0 0 1 0 0 0	0 0 1 0 0 0 1 0	1 1 1 0 1 1 1 1	1 1 1 1 0 0 0 0	1 1 0 0 0 0 0 0	1 0 1 0 0 0 1 1	0 0 0 0 1 0 0 0
DAT_input_output$_{\mathrm{Hex}}$	0x9A	0x88	0x22	0xEF	0xF0	0xC0	0xA3	0x08

Table 172: Example uint64

Decimal	-113							
Binary	1	0	0	0	1	1	1	1
Hex	0x8F							
	MSB							LSB

Position	n							
DAT_input_output_Bin	1	0	0	0	1	1	1	1
DAT_input_output_Hex	0x8F							

Table 173: Example int8

Decimal	-4963															
Binary	1	1	1	0	1	1	0	0	1	0	0	1	1	1	0	1
Hex	0xEC								0x9D							
	MSB															LSB

Position	n								n + 1							
DAT_input_output_Bin	1	0	0	1	1	1	0	1	1	1	1	0	1	1	0	0
DAT_input_output_Hex	0x9D								0xEC							

Table 174: Example int16

Decimal	-89498498																															
Binary	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0
Hex	0xFA								0xAA								0x5C								0x7E							
	MSB																															LSB

Position	N								n + 1								n + 2								n + 3							
DAT_input_output_Bin	0	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	1	0	1	1	1	1	1	0	1	0
DAT_input_output_Hex	0x7E								0x5C								0xAA								0xFA							

Table 175: Example int32

Decimal	-8789498491988154686							
Binary	1 0 0 0 0 1 1 0	0 0 0 0 0 1 0 1	0 1 1 0 1 1 0 1	1 0 1 1 0 0 0 0	1 0 1 1 0 1 1 1	0 0 1 1 0 1 1 0	1 1 1 0 0 1 1 0	1 1 0 0 0 0 1 0
Hex	0x86	0x05	0x6D	0xB0	0xB7	0x36	0xE6	0xC2
	MSB							LSB

Position	n	n + 1	n+ 2	n + 3	n + 4	n + 5	n + 6	n + 7
DAT_input_output$_{\text{Bin}}$	1 1 0 0 0 0 1 0	1 1 1 0 0 1 1 0	0 0 1 1 0 1 1 0	1 0 1 1 0 1 1 1	1 0 1 1 0 0 0 0	0 1 1 0 1 1 0 1	0 0 0 0 0 1 0 1	1 0 0 0 0 1 1 0
DAT_input_output$_{\text{Hex}}$	0xC2	0xE6	0x36	0xB7	0xB0	0x6D	0x05	0x86

Table 176: Example int64

Decimal	7256.2568359375			
Binary	0 1 0 0 0 1 0 1	1 1 1 0 0 0 1 0	1 1 0 0 0 0 1 0	0 0 0 0 1 1 1 0
Hex	0x45	0xE2	0xC2	0x0E
	MSB			LSB

Position	n	n + 1	n + 2	n + 3
DAT_input_output$_{\text{Bin}}$	0 0 0 0 1 1 1 0	1 1 0 0 0 0 1 0	1 1 1 0 0 0 1 0	0 1 0 0 0 1 0 1
DAT_input_output$_{\text{Hex}}$	0x0E	0xC2	0xE3	0x45

Table 177: Example float32

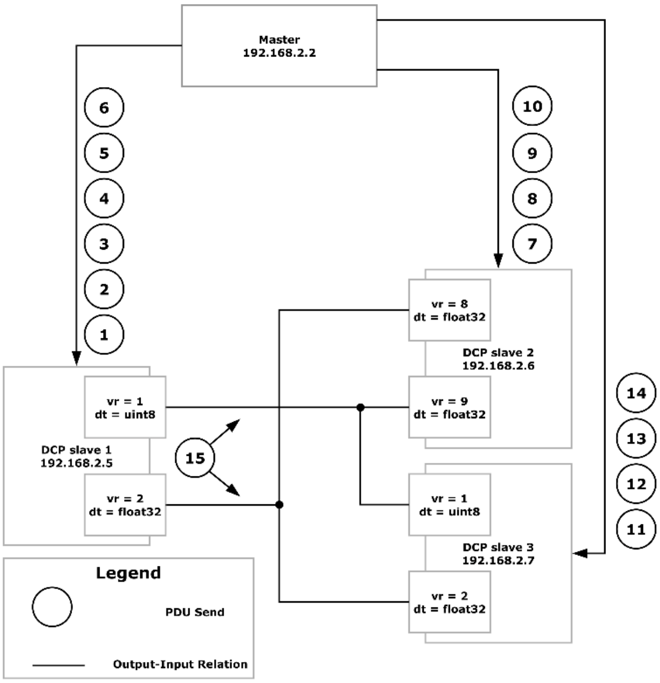
Decimal	46.42829231507700882275457843206822872161865234375							
Binary	0 1 0 0 0 0 0 0	0 1 0 0 0 1 1 1	0 0 1 1 0 1 1 0	1 1 0 1 0 0 1 0	0 1 0 0 1 0 0 0	0 0 1 1 0 0 0 1	0 0 1 1 0 0 0 1	0 0 1 0 0 0 1 1
Hex	0x40	0x47	0x36	0xD2	0x48	0x57	0x31	0x23
	MSB							LSB

Position	n	n + 1	n+ 2	n + 3	n + 4	n + 5	n + 6	n + 7
DAT_input_output$_{\text{Bin}}$	0 0 1 0 0 0 1 1	0 0 1 1 0 0 0 1	0 0 1 1 0 0 0 1	0 1 0 0 1 0 0 0	1 1 0 1 0 0 1 0	0 0 1 1 0 1 1 0	0 1 0 0 0 1 1 1	0 1 0 0 0 0 0 0
DAT_input_output$_{\text{Hex}}$	0x23	0x31	0x57	0x48	0xD2	0x36	0x47	0x40

Table 178: Example float64

D. Example: Data Exchange

The following Figure 33 shows the intended sequence of necessary steps for configuration roll-out using native DCP (UDP over IPv4).



1 CFG_set_scope
type_id = 0x2B
pdu_seq_id = 1
receiver = 1
data_id = 0
scope = RUNNING

2 CFG_config_output
type_id = 0x23
pdu_seq_id = 2
receiver = 1
data_id = 0
pos = 0
source_vr = 1

3 CFG_config_output
type_id = 0x23
pdu_seq_id = 3
receiver = 1
data_id = 0
pos = 0
source_vr = 2

4 CFG_set_target_network_information
type_id = 0x25
pdu_seq_id = 4
receiver = 1
data_id = 0
transport_protocol = UDP
target_ip_address = 192.168.2.6
target_port=60000

5 CFG_set_target_network_information
type_id = 0x25
pdu_seq_id = 5
receiver = 1
data_id = 0
transport_protocol = UDP
target_ip_address = 192.168.2.7
target_port=60000

6 CFG_set_steps
type_id = 0x21
pdu_seq_id = 6
receiver = 1
steps = 1
data_id = 0

7 CFG_set_scope
type_id = 0x2B
pdu_seq_id = 1
receiver = 2
data_id = 0
scope = RUNNING

8 CFG_config_input
type_id = 0x22
pdu_seq_id = 2
receiver = 2
data_id = 0
pos = 0
target_vr = 2
source_data_type = uint8

9 CFG_config_input
type_id = 0x22
pdu_seq_id = 3
receiver = 2
data_id = 0
pos = 0
target_vr = 2
source_data_type = float32

10 CFG_set_source_network_information
type_id = 0x26
pdu_seq_id = 4
receiver = 2
data_id = 0
transport_protocol = UDP
target_ip_address = 192.168.2.6
target_port=60000

11 CFG_set_scope
type_id = 0x2B
pdu_seq_id = 1
receiver = 3
data_id = 0
scope = RUNNING

12 CFG_config_input
type_id = 0x22
pdu_seq_id = 2
receiver = 3
data_id = 0
pos = 0
target_vr = 2
source_data_type = uint8

13 CFG_config_input
type_id = 0x22
pdu_seq_id = 3
receiver = 3
data_id = 0
pos = 0
target_vr = 2
source_data_type = float32

14 CFG_set_source_network_information
type_id = 0x26
pdu_seq_id = 4
receiver = 3
data_id = 0
transport_protocol = UDP
target_ip_address = 192.168.2.7
target_port=60000

15 DAT_input_output
type_id = 0x2B
pdu_seq_id = 0
data_id = 0
payload =

uint8	uint8

Note: For better legibility all values are in human readable format. On the communication medium these values are transported as defined in this specification.

Figure 33: Example configuration roll-out

E. Validation of PDU Sequence ID

This section highlights a possible way to check the pdu_seq_id for integrity. The pdu_seq_id may be considered valid if the following holds:

$$\left(pdu\_seq\_id_{cur}\text{- } pdu\_seq\_id_{prev} - 1\right) mod\ (2^{16}) < \frac{2^{16}}{2}$$

Equation 1: pdu_seq_id integrity check.

where pdu_seq_id$_{prev}$ is the last pdu_seq_id which passed this check. The initial value of pdu_seq_id$_{prev}$ is defined as the first processed pdu_seq_id, whose associated PDU is accepted without passing Equation 1.

Example: pdu_seq_id$_{cur}$ = 233, pdu_seq_id$_{prev}$ = 234, (233-234) mod (65536) = -1 mod 65536 = 65535, therefore this PDU is rejected.

Explanation:
It needs to be ensured that the pdu_seq_id of a received PDU belongs to a newer PDU than the pdu_seq_id of the previously received one. The simple smaller operator (<) is not sufficient, because of the following property of the pdu_seq_id:

- After reaching $2^{16} - 1$ the pdu_seq_id is reset to 0. Therefore, the pdu_seq_id 1 can be newer than 65535.

This has the consequence that the exact distance between two pdu_seq_id cannot be calculated.

Example:
If the previous pdu_seq_id was 5 and the received pdu_seq_id is 4 than the distance could be one of {-1, -1 + 2^{16}, -1 + $2 \cdot 2^{16}$, $-1\ + 3 \cdot 2^{16}, \ldots$ }. Therefore, a heuristic must be used. To determine if a received pdu_seq_id is valid the following distinction is made:

- The next $\frac{2^{16}}{2}$ increases from the previous received pdu_seq_id are valid
- The previous $\frac{2^{16}}{2} - 1$ values, aswell as the last received pdu_seq_id are invalid.

To calculate how many increases i are necessary to reach the current pdu_seq_id from the previous one, the following formula can be used:

$$i = \left(pdu\_seq\_id_{cur}\text{- } pdu\_seq\_id_{prev} - 1\right) mod\ (2^{16})$$

Combined with the above, this means that a pdu_seq_id is valid if Equation 1 holds.

This heuristic fails if $\frac{2^{16}}{2}$ PDU losses occur subsequently. After that a valid pdu_seq_id would be identified as invalid. After $\frac{2^{16}}{2}$ falsely rejected pdu_seq_ids the heuristic would accept the upcoming pdu_seq_ids again.

Assume a that the package loss is independent from each other and has a probability of 99%. Then the probability that this heuristic fails equals $9.4 \cdot 10^{-144}$. For systematic PDU loss, like network failure, the simulation must be stopped and restarted anyway.