SMPTE RDD 38:2016

SMPTE REGISTERED DISCLOSURE DOCUMENT

Networked Device Control Protocol — Message Data Structure and Method of Communication

The attached document is a Registered Disclosure Document prepared by the proponent identified below. It has been examined by the appropriate SMPTE Technology Committee and is believed to contain adequate information to satisfy the objectives defined in the Scope, and to be technically consistent.

This document is NOT a Standard, Recommended Practice or Engineering Guideline, and does NOT imply a finding or representation of the Society.

Errors in this document should be reported to the proponent identified below, with a copy to eng@smpte.org. All other inquiries in respect of this document, including inquiries as to intellectual property requirements that may be attached to use of the disclosed technology, should be addressed to the proponent identified below.

Proponent contact information:

Satoshi Katsuo
Sony Corporation
4-14-1 Asahi-cho, Atsugi
Kanagawa, 243-0014
Japan

Email: Satoshi.Katsuo@sony.com

Approved
June 30, 2016

Table of Contents

Page

Introduction

The phrase “IP control” covers a wide range of control functions via IP networks, from business level control (control at workflow level) to device level control (control at function level).

As for the latter, it has now become common practice to control a product using a variety of controller types including portable computers, smartphones and/or tablets via wireless IP networks.

In such situations, even in an IP network environment there is a requirement to realize high speed control/response equivalent to using a conventional RS422/9-pin control device. This RDD provides a lightweight and efficient control protocol specification for this purpose.

Specifically, it conforms to MessagePack and MessagePack RPC specifications, with some added definitions required for device control. These include an efficient method of message communication, and use of available transport protocols that take security into account.

See Annex A for a URL to an open source reference implementation.

1 Scope

This RDD provides the following specification for a protocol layer to control networked devices efficiently in an IP network environment.

- Serialized object format
- Message format
- Message sequence
- Transport protocol
- Security

2 Related Documents and URLs

IETF RFC 2617, HTTP Authentication: Basic and Digest Access Authentication

IETF RFC 3629, UTF-8, a transformation format of ISO 10646

IETF RFC 5246, The Transport Layer Security (TLS) Protocol Version 1.2

IETF RFC 6455, The WebSocket Protocol

The Unicode Consortium, The Unicode Standard

IEEE 754, Standard for Floating Point Arithmetic

MessagePack, http://msgpack.org/

MessagePack RPC, https://github.com/msgpack-rpc/

3 Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Big-endian byte order

The most significant byte of words is stored at a particular memory address and subsequent bytes are stored in the following higher memory addresses, the least significant byte thus being stored at the highest memory address. It is also called network order in the IETF RFC documents.

3.2 Deserialization

The process of translating a series of bytes into application data structures or object states.

3.3 Least Significant Byte (LSB)

The byte in that position of a multi-byte number which has the least potential value.

3.4 Most Significant Byte (MSB)

The byte in that position of a multi-byte number which has the greatest potential value.

3.5 Serialization

The process of translating data structures or object state into a series of bytes that can be stored and reconstructed later in the same or another computer environment.

4 Serialized Object Formats

4.1 General

This section describes data formats of the serialized object which have an arbitrary type by a binary string. Since serialized data has both type information and a value, an application object can be reconstructed from it.

Table 1 shows the defined types in serialized object formats. In addition to basic types defined by most programming languages, a variable length binary/string data type and container types are also defined.

In order to minimize the post-serialization data size, the type can be set by the actual value at the time of serialization instead of strictly serializing the type as declared by the programming language. For example, even if a type is declared as 64-bit unsigned integer by the programming language, if the value is 250, the data will be serialized into an 8-bit unsigned integer. As a result, the post-serialization data size is 7 bytes shorter than with serialization into a 64-bit unsigned integer.

An open source reference implementation described in Annex A supports all data formats defined in this section, and also has the size reduction capability described above.

In the following sections, bits in a byte are labeled 7 through 0. The most significant bit is bit number 7, and the least significant bit is bit number 0.

Table 1 – Defined data types in serialized object formats

Type	First byte (in binary)	First byte (in hex)	Details
7-bit positive integer	0xxxxxxx	0x00 - 0x7f	See 4.4.2.
map in 4-bit object number	1000xxxx	0x80 - 0x8f	See 4.9.2.
array in 4-bit object number	1001xxxx	0x90 - 0x9f	See 4.8.2.
UTF-8 string in 5-bit length	101xxxxx	0xa0 - 0xbf	See 4.6.2.
Nil	11000000	0xc0	See 4.2.
(never used)	11000001	0xc1	
False	11000010	0xc2	See 4.3.2.
True	11000011	0xc3	See 4.3.3.
binary in 8-bit length	11000100	0xc4	See 4.7.2.
binary in 16-bit length	11000101	0xc5	See 4.7.3.
binary in 32-bit length	11000110	0xc6	See 4.7.4.
user defined type in 8-bit length	11000111	0xc7	See 4.10.2.
user defined type in 16-bit length	11001000	0xc8	See 4.10.3.
user defined type in 32-bit length	11001001	0xc9	See 4.10.4.
IEEE 754 single precision floating point number	11001010	0xca	See 4.5.2.
IEEE 754 double precision floating point number	11001011	0xcb	See 4.5.3.
8-bit unsigned integer	11001100	0xcc	See 4.4.4.
16-bit unsigned integer	11001101	0xcd	See 4.4.5.
32-bit unsigned integer	11001110	0xce	See 4.4.6.
64-bit unsigned integer	11001111	0xcf	See 4.4.7.
8-bit signed integer	11010000	0xd0	See 4.4.8.
16-bit signed integer	11010001	0xd1	See 4.4.9.
32-bit signed integer	11010010	0xd2	See 4.4.10.
64-bit signed integer	11010011	0xd3	See 4.4.11.
8-bit user defined type	11010100	0xd4	See 4.10.2.
16-bit user defined type	11010101	0xd5	See 4.10.3.
32-bit user defined type	11010110	0xd6	See 4.10.4.
64-bit user defined type	11010111	0xd7	See 4.10.5.
128-bit user defined type	11011000	0xd8	See 4.10.6.
UTF-8 string in 8-bit length	11011001	0xd9	See 4.6.3.
UTF-8 string in 16-bit length	11011010	0xda	See 4.6.4.
UTF-8 string in 32-bit length	11011011	0xdb	See 4.6.5.
array in 16-bit object number	11011100	0xdc	See 4.8.3.
array in 32-bit object number	11011101	0xdd	See 4.8.4.
map in 16-bit object number	11011110	0xde	See 4.9.3.
map in 32-bit object number	11011111	0xdf	See 4.9.4.
5-bit negative integer	111xxxxx	0xe0 - 0xff	See 4.4.3.

4.2 Nil

Table 2 shows a nil. In most programming language, nil refers to an empty set or a list containing no entries. Nil is also used when a value is not assigned to a variable.

Table 2 – nil

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc0)							
	1	1	0	0	0	0	0	0

4.3 Boolean

4.3.1 General

Boolean data types usually have two values such as true or false.

4.3.2 false

Table 3 shows a false. It is represented by 0xc2 in hexadecimal.

Table 3 – false

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc2)							
	1	1	0	0	0	0	1	0

4.3.3 true

Table 4 shows a true. It is represented by 0xc3 in hexadecimal.

Table 4 – true

	7	6	5	4	3	2	1	0
byte 1	Type (0xc3)							
	1	1	0	0	0	0	1	1

4.4 Integer

4.4.1 General

An integer value is stored big-endian order in the object serialization formats.

4.4.2 7-bit positive integer

Table 5 shows the format for a positive integer value. The type field is the most significant bit of byte 1 and is stored as 0. A positive integer value should be stored as this format if the value is less than 128.

Table 5 – 7-bit positive integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type	Value (0..127)						
	0	X	X	X	X	X	X	X

4.4.3 5-bit negative integer

Table 6 shows the format for a negative integer value. The type field in byte 1 is stored as 111 in binary. A negative integer should be stored as this format if the value is larger than -32.

Table 6 – 5-bit negative integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type			Value (-31..-1)				
	1	1	1	X	X	X	X	X

4.4.4 8-bit unsigned integer

Table 7 shows the format for an 8-bit unsigned integer value. The type field in byte 1 is stored as 0xcc in hexadecimal. The value field in byte 2 is stored as an 8-bit unsigned integer value. An unsigned integer value should be stored as this format if the value is within the max value of an 8-bit unsigned integer.

Table 7 – 8-bit unsigned integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xcc)							
	1	1	0	0	1	1	0	0
byte 2	Value							

4.4.5 16-bit unsigned integer

Table 8 shows the format for a 16-bit unsigned integer value. The type field in byte 1 is stored as 0xcd in hexadecimal. The value field is stored as a 16-bit unsigned integer value in big-endian order. An unsigned integer value should be stored as this format if the value is within the max value of a 16-bit unsigned integer.

Table 8 – 16-bit unsigned integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xcd)							
	1	1	0	0	1	1	0	1
byte 2	Value (MSB)							
byte 3	Value (LSB)							

4.4.6 32-bit unsigned integer

Table 9 shows the format for a 32-bit unsigned integer value. The type field in byte 1 is stored as 0xce in hexadecimal. The value field is stored as a 32-bit unsigned integer value in big-endian order. An unsigned integer value should be stored as this format if the value is within the max value of a 32-bit unsigned integer.

Table 9 – 32-bit unsigned integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xce)							
	1	1	0	0	1	1	1	0
byte 2	Value (MSB)							
byte 3	Value (cont.)							
byte 4	Value (cont.)							
byte 5	Value (LSB)							

4.4.7 64-bit unsigned integer

Table 10 shows the format for a 64-bit unsigned integer value. The type field in byte 1 is stored as 0xcf in hexadecimal. The value field is stored as a 64-bit unsigned integer value in big-endian order. An unsigned integer value should be stored as this format if the value is within the max value of a 64-bit unsigned integer.

Table 10 – 64-bit unsigned integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xcf)							
	1	1	0	0	1	1	1	1
byte 2	Value (MSB)							
byte 3	Value (cont.)							
byte 4	Value (cont.)							
byte 5	Value (cont.)							
byte 6	Value (cont.)							
byte 7	Value (cont.)							
byte 8	Value (cont.)							
byte 9	Value (LSB)							

4.4.8 8-bit signed integer

Table 11 shows the format for an 8-bit signed integer value. The type field in byte 1 is stored as 0xd0 in hexadecimal. The value field is stored as an 8-bit signed integer value. A signed integer value should be stored as this format if the value is within the range of an 8-bit signed integer.

Table 11 – 8-bit signed integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd0)							
	1	1	0	1	0	0	0	0
byte 2	Value							

4.4.9 16-bit signed integer

Table 12 shows the format for a 16-bit signed integer value. The type field in byte 1 is stored as 0xd1 in hexadecimal. The value field is stored as a 16-bit signed integer value in big-endian order. A signed integer value should be stored as this format if the value is within the range of a 16-bit signed integer.

Table 12 – 16-bit singed integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd1)							
	1	1	0	1	0	0	0	1
byte 2	Value (MSB)							
byte 3	Value (LSB)							

4.4.10 32-bit signed integer

Table 13 shows the format for a 32-bit signed integer value. The type field in byte 1 is stored as 0xd2 in hexadecimal. The value field is stored as a 32-bit signed integer value in big-endian order. A signed integer value should be stored this format if the value is within the range of a 32-bit signed integer.

Table 13 – 32-bit signed integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd2)							
	1	1	0	0	1	1	1	0
byte 2	Value (MSB)							
byte 3	Value (cont.)							
byte 4	Value (cont.)							
byte 5	Value (LSB)							

4.4.11 64-bit signed integer

Table 14 shows the format for a 64-bit signed integer value. The type field in byte 1 is stored as 0xd3 in hexadecimal. The value field is stored as a 64-bit signed integer value in big-endian order. A signed integer value should be stored this format if the value is within the range of a 64-bit signed integer.

Table 14 – 64-bit signed integer format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd3)							
	1	1	0	1	0	0	1	1
byte 2	Value (MSB)							
byte 3	Value (cont.)							
byte 4	Value (cont.)							
byte 5	Value (cont.)							
byte 6	Value (cont.)							
byte 7	Value (cont.)							
byte 8	Value (cont.)							
byte 9	Value (LSB)							

4.5 IEEE 754 Floating Point Numbers

4.5.1 General

This specification supports IEEE 754 single precision floating point number and IEEE 754 double precision floating point number.

4.5.2 Single precision floating point number

Table 15 shows the format for the value of an IEEE 754 single precision floating point number. The type field in byte 1 is stored as 0xca in hexadecimal. The most significant bit in byte 2 stores the sign value: 0 for a positive number and 1 for a negative number. The exponent field is stored as an 8-bit exponent value. The fraction field in the format is stored as a 23-bit fraction value.

Table 15 – IEEE 754 single precision floating point number format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xca)							
	1	1	0	0	1	0	1	0
byte 2	Sign	Exponent						
byte 3	Exponent (cont.)	Fraction						
byte 4	Fraction (cont.)							
byte 5	Fraction (cont.)							

4.5.3 Double precision floating point number

Table 16 shows the format for the value of an IEEE 754 double precision floating point number. The type field in byte 1 is stored as 0xcb in hexadecimal. The most significant bit in byte 2 stores the sign value: 0 for a positive number and 1 for a negative number. The exponent field is stored as an 11-bit exponent value. The fraction field in the format is stored as 52-bit fraction value.

Table 16 – IEEE 754 double precision floating point number format

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xcb)							
	1	1	0	0	1	0	1	1
byte 2	Sign	Exponent						
byte 3	Exponent (cont.)				Fraction			
byte 4	Fraction (cont.)							
byte 5	Fraction (cont.)							
byte 6	Fraction (cont.)							
byte 7	Fraction (cont.)							
byte 8	Fraction (cont.)							
byte 9	Fraction (cont.)							

4.6 UTF-8 String

4.6.1 General

In this format, strings are stored as UTF-8 encoded string. UTF-8 is an efficient encoding of Unicode characters that optimizes the encoding of ASCII characters. Four types of formats exist according to the string length.

4.6.2 5-bit string length

Table 17 shows the format for a string with a 5-bit length field. The type field in byte 1 is stored as 101 in binary. The string length field in byte 1 is stored as a 5-bit unsigned integer value. Strings should be stored as this format if the string length is within the max value of a 5-bit unsigned integer.

Table 17 – UTF-8 string format in 5-bit string length

Bit	7	6	5	4	3	2	1	0
byte 1	Type			String length				
	1	0	1	X	X	X	X	X
byte 2 …	UTF-8 string, if length > 0.							

4.6.3 8-bit string length

Table 18 shows the format for a string with an 8-bit length field. The type field in byte 1 is stored as 0xd9 in hexadecimal. The string length field is stored as an 8-bit unsigned integer value. Strings should be stored this format if the string length is within the max value of an 8-bit unsigned integer.

Table 18 – UTF-8 string format in 8-bit string length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd9)							
	1	1	0	1	1	0	0	1
byte 2	String length							
byte 3 …	UTF-8 string, if length > 0.							

4.6.4 16-bit string length

Table 19 shows the format for a string with a 16-bit length field. The type field in byte 1 is stored as 0xda in hexadecimal. The string length field is stored as a 16-bit unsigned integer value in big-endian order. Strings should be stored this format if the string length is within the max value of a 16-bit unsigned integer.

Table 19 – UTF-8 string format in 16-bit string length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xda)							
	1	1	0	1	1	0	1	0
byte 2	String length (MSB)							
byte 3	String length (LSB)							
byte 4 …	UTF-8 string, if length > 0.							

4.6.5 32-bit string length

Table 20 shows the format for a string with a 32-bit length field. The type field in byte 1 is stored as 0xdb in hexadecimal. The string length field is stored as a 32-bit unsigned integer value in big-endian order. Strings should be stored as this format if the string length is within max value of a 32-bit unsigned integer.

Table 20 – UTF-8 string format in 32-bit string length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xdb)							
	1	1	0	0	1	1	1	0
byte 2	String length (MSB)							
byte 3	String length (cont.)							
byte 4	String length (cont.)							
byte 5	String length (LSB)							
byte 6 …	UTF-8 string, if length > 0.							

4.7 Binary Data

4.7.1 General

Three types of formats exist according to the binary data length.

4.7.2 8-bit binary data length

Table 21 shows the format for binary data with an 8-bit length field. The type field in byte 1 is stored as 0xc4 in hexadecimal. The binary data length field is stored as an 8-bit unsigned integer value. Binary data should be stored as this format if the binary data length is within the max value of an 8-bit unsigned integer.

Table 21 – Binary data format in 8-bit binary data length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc4)							
	1	1	0	0	0	1	0	0
byte 2	Binary length							
byte 3 …	Binary data, if length > 0.							

4.7.3 16-bit binary data length

Table 22 shows the format for binary data with a 16-bit length field. The type field in byte 1 is stored as 0xc5 in hexadecimal. The binary data length field is stored as a 16-bit unsigned integer value in big-endian order. Binary data should be stored as this format if the binary data length is within the max value of a 16-bit unsigned integer.

Table 22 – Binary data format in 16-bit binary data length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc5)							
	1	1	0	0	0	1	0	1
byte 2	Binary length (MSB)							
byte 3	Binary length (LSB)							
byte 4 …	Binary data, if length > 0							

4.7.4 32-bit binary data length

Table 23 shows the format for binary data with a 32-bit length field. The type field in byte 1 is stored as 0xc6 in hexadecimal. The binary data length field is stored as a 32-bit unsigned integer value in big-endian order. Binary data should be stored as this format if the binary data length is within the max value of a 32-bit unsigned integer.

Table 23 – Binary data format in 32-bit binary data length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc6)							
	1	1	0	0	0	1	1	0
byte 2	Binary length (MSB)							
byte 3	Binary length (cont.)							
byte 4	Binary length (cont.)							
byte 5	Binary length (LSB)							
byte 6 …	Binary data, if length > 0.							

4.8 Array

4.8.1 General

An array is an indexed container of the object defined in Table 1. Indexed indicates that the objects are numbered from 0. An array itself is also the object. Three types of formats exist according to the number of objects.

4.8.2 4-bit object number

Table 24 shows the format for an array with a 4-bit object number field. The type field in byte 1 is stored as 1001 in binary. The object number field in byte 1 is stored as a 4-bit unsigned integer value. Array data should be stored as this format if the object number is within the max value of a 4-bit unsigned integer.

Table 24 – Array format in 4-bit object number

Bit	7	6	5	4	3	2	1	0
byte 1	Type				Object number			
	1	0	0	1	X	X	X	X
byte 2 …	Object, if number > 0.							

4.8.3 16-bit object number

Table 25 shows the format for an array with a 16-bit object number field. The type field in byte 1 is stored as 0xdc in hexadecimal. The object number field is stored as a 16-bit unsigned integer value in big-endian order. Array data should be stored as this format if the object number is within the max value of a 16-bit unsigned integer.

Table 25 – Array format in 16-bit object number

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xdc)							
	1	1	0	1	1	1	0	1
byte 2	Object number (MSB)							
byte 3	Object number (LSB)							
byte 4 …	Object, if number > 0.							

4.8.4 32-bit object number

Table 26 shows the format for an array with a 32-bit object number field. The type field in byte 1 is stored as 0xdd in hexadecimal. The object number field is stored as a 32-bit unsigned integer value in big-endian order. Array data should be stored as this format if the object number is within the max value of a 16-bit unsigned integer.

Table 26 – Array format in 32-bit object number

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xdd)							
	1	1	0	1	1	1	1	0
byte 2	Object number (MSB)							
byte 3	Object number (cont.)							
byte 4	Object number (cont.)							
byte 5	Object number (LSB)							
byte 6 …	Object, if number > 0.							

4.9 Map

4.9.1 General

A map is an associative container for storing elements formed by combination of a key value and a mapped value. Three types of formats exist according to the number of key-value pairs.

4.9.2 4-bit element number

Table 27 shows the format for a map with a 4-bit element number field. The type field in byte 1 is stored as 1000 in binary. The element number field is stored as a 4-bit unsigned integer value. Map data should be stored as this format if the element number is within max value of a 4-bit unsigned integer.

Table 27 – Map format in 4-bit element number

Bit	7	6	5	4	3	2	1	0
byte 1	Type				Element number			
	1	0	0	0	X	X	X	X
byte 2	Key object 1, if number > 0.							
byte 3	Value object 1, if number > 0.							
byte 4	Key object 2, if number > 1							
byte 5 …	Value object 2, if number > 1							

4.9.3 16-bit element number

Table 28 shows the format for a map with a 16-bit element number field. The type field in byte 1 is stored as 0xde in hexadecimal. The element number field is stored as a 16-bit unsigned integer value in big-endian order. Map data should be stored as this format if the element number is within the max value of a 16-bit unsigned integer.

Table 28 – Map format in 16-bit element number

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xde)							
	1	1	0	1	1	1	1	0
byte 2	Element number (MSB)							
byte 3	Element number (LSB)							
byte 4 …	Key object 1, if number > 0.							
…	Value object 1, if number > 0.							
…	Key object 2, if number > 1							
…	Value object 2, if number > 1							

4.9.4 32-bit element number

Table 29 shows the format for a map with a 32-bit element number field. The type field in byte 1 is stored as 0xde in hexadecimal. The element number field is stored as a 32-bit unsigned integer value in big-endian order. Map data should be stored as this format if the element number is within the max value of a 32-bit unsigned integer.

Table 29 – Map format in 32-bit element number

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xdf)							
	1	1	0	1	1	1	1	1
byte 2	Element number (MSB)							
byte 3	Element number (cont.)							
byte 4	Element number (cont.)							
byte 5	Element number (LSB)							
byte 6 …	Key object 1, if number > 0.							
…	Value object 1, if number > 0.							
…	Key object 2, if number > 1							
…	Value object 2, if number > 1							

4.10 User Defined Types

4.10.1 General

This specification also provides extensible formats for user defined types or programming language specific types.

4.10.2 8-bit data

Table 30 shows the format for a user defined type in 8 bits. The type field in byte 1 is stored as 0xd4 in hexadecimal. The user defined type field in byte 2 is stored as a user defined type number.

Table 30 – User defined type format in 8-bit data

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd4)							
	1	1	0	1	0	1	0	0
byte 2	User defined type							
byte 3	Data							

4.10.3 16-bit data

Table 31 shows the format for a user defined type in 16 bits. The type field in byte 1 is stored as 0xd5 in hexadecimal. The user defined type field in byte 2 is stored as a user defined type number.

Table 31 – User defined type format in 16-bit data

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd5)							
	1	1	0	1	0	1	0	0
byte 2	User defined type							
byte 3	Data							
byte 4	Data (cont.)							

4.10.4 32-bit data

Table 32 shows the format for a user defined type in 32 bits. The type field in byte 1 is stored as 0xd6 in hexadecimal. The user defined type field in byte 2 is stored as a user defined type number.

Table 32 – User defined type format in 32-bit data

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd6)							
	1	1	0	1	0	1	1	0
byte 2	User defined type							
byte 3	Data							
byte 4	Data (cont.)							
byte 5	Data (cont.)							
byte 6	Data (cont.)							

4.10.5 64-bit data

Table 33 shows the format for a user defined type in 64 bits. The type field in byte 1 is stored as 0xd7 in hexadecimal. The user defined type field in byte 2 is stored as a user defined type number.

Table 33 – User defined type format in 64-bit data

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd7)							
	1	1	0	1	0	1	1	1
byte 2	User defined type							
byte 3	Data							
byte 4	Data (cont.)							
byte 5	Data (cont.)							
byte 6	Data (cont.)							
byte 7	Data (cont.)							
byte 8	Data (cont.)							
byte 9	Data (cont.)							
byte 10	Data (cont.)							

4.10.6 128-bit data

Table 34 shows the format for a user defined type in 128 bits. The type field in byte 1 is stored as 0xd8 in hexadecimal. The user defined type field in byte 2 is stored as a user defined type number.

Table 34 – User defined type format in 128-bit data

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xd8)							
	1	1	0	1	1	0	0	0
byte 2	User defined type							
byte 3	Data							
byte 4	Data (cont.)							
byte 5	Data (cont.)							
byte 6	Data (cont.)							
byte 7	Data (cont.)							
byte 8	Data (cont.)							
byte 9	Data (cont.)							
byte 10	Data (cont.)							
byte 11	Data (cont.)							
byte 12	Data (cont.)							
byte 13	Data (cont.)							
byte 14	Data (cont.)							
byte 15	Data (cont.)							
byte 16	Data (cont.)							
byte 17	Data (cont.)							
byte 18	Data (cont.)							

4.10.7 8-bit length

Table 35 shows the format for a user defined type with an 8-bit data length field. The type field in byte 1 is stored as 0xc7 in hexadecimal. The data length field in byte 2 is stored as an 8-bit unsigned integer value.

Table 35 – User defined type format in 8-bit length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc7)							
	1	1	0	0	0	1	0	0
byte 2	Data length							
byte 3	User defined type							
byte 4 …	Data, if length > 0.							

4.10.8 16-bit length

Table 36 shows the format for a user defined type with a 16-bit data length field. The type field in byte 1 is stored as 0xc8 in hexadecimal. The data length field is stored as a 16-bit unsigned integer value in big-endian order.

Table 36 – User defined type format in 16-bit length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc8)							
	1	1	0	0	1	0	0	0
byte 2	Data length (MSB)							
byte 3	Data length (LSB)							
byte 4	User defined type							
byte 5 …	Data, if length > 0.							

4.10.9 32-bit length

Table 37 shows the format for a user defined type with a 32-bit data length field. The type field in byte 1 is stored as 0xc9 in hexadecimal. The data length field is stored as a 32-bit unsigned integer value in big-endian order.

Table 37 – User defined type format in 32-bit length

Bit	7	6	5	4	3	2	1	0
byte 1	Type (0xc9)							
	1	1	0	0	1	0	0	0
byte 2	Data length (MSB)							
byte 3	Data length (cont.)							
byte 4	Data length (cont.)							
byte 5	Data length (LSB)							
byte 6	User defined type							
byte 7 …	Data, if length > 0.							

5 Message Formats

5.1 General

This section describes three kinds of message formats: a request message, a response message and a notify message. Each message is serialized into an array format in a manner described in Section 4.

5.2 Request Message

A request message shall consist of an array containing the four elements specified in Table 38.

Table 38 – The elements of a request message

Element	Description
type	An integer that represents the type of message. This field shall contain “0” for a request message.
msgid	A 32-bit unsigned integer. This field shall contain the sequence number of a message. The value shall be reset to “0” if the sequence number exceeds the maximum value of 2^{32}-1.
method	A character string. This field shall contain the method name. The method name depends on the command definition which is beyond the scope of this document.
params	An object array or an object defined in Table 1. An object array shall be applied when compatibility with MessagePack RPC is required. This field shall contain the argument(s) of method. The applicable object depends on the command definition which is beyond the scope of this document.

Table 39 shows an example of a request message. The method name is ‘Hello’, and the element params is an array consisting of an integer and a string.

Table 39 – An example of a request message

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (4)			
	1	0	0	1	0	1	0	0
byte 2	Type – Request Message (0)							
	0	0	0	0	0	0	0	0
byte 3	Type (0xce) – unsigned integer							
	1	1	0	0	1	1	1	0
byte 4	Msgid (MSB)							
byte 5	Msgid (cont.)							
byte 6	Msgid (cont.)							
byte 7	Msgid (LSB)							
byte 8	Type - string			String length				
	1	0	1	0	0	1	0	1
byte 9	‘H’							
byte 10	‘e’							
byte 11	‘l’							
byte 12	‘l’							
byte 13	‘o’							
byte 14	Type - array				Object number (2)			
	1	0	0	1	0	0	1	0
byte 15	0	0	0	0	0	0	1	1
byte 16	Type - string			String length (5)				
	1	0	1	0	0	1	0	1
byte 17	‘P’							
byte 18	‘a’							
byte 19	‘r’							
byte 20	‘a’							
byte 21	‘m’							

Table 40 shows an example of a request message with no parameters. The method name is ‘Hello’, and the element params is an empty array.

Table 40 – An example of a request message with no parameters

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (4)			
	1	0	0	1	0	1	0	0
byte 2	Type – Request Message (0)							
	0	0	0	0	0	0	0	0
byte 3	Type (0xce) – unsigned integer							
	1	1	0	0	1	1	1	0
byte 4	Msgid (MSB)							
byte 5	Msgid (cont.)							
byte 6	Msgid (cont.)							
byte 7	Msgid (LSB)							
byte 8	Type - string			String length				
	1	0	1	0	0	1	0	1
byte 9	‘H’							
byte 10	‘e’							
byte 11	‘l’							
byte 12	‘l’							
byte 13	‘o’							
byte 14	Type - array				Object number (0)			
	1	0	0	1	0	0	0	0

5.3 Response Message

A response message shall consist of an array containing the four elements specified in Table 41.

Table 41 – The elements of a response message

Element	Description
type	An integer that represents the type of message. This field shall contain “1” for a response message.
msgid	A 32-bit unsigned integer. This field shall contain the sequence number of the corresponding request message.
error	An object defined in Table 1. This field shall contain the result of a failed process. The value shall be “nil” if a process was successful.
result	An object defined in Table 1. This field shall contain the result of a successful process. The value shall be “nil” if a process failed or a case of void return.

Table 42 shows an example of a success response message with an unsigned integer result.

Table 42 – An example of a success response message with an unsigned integer result

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (4)			
	1	0	0	1	0	1	0	0
byte 2	Type – Response Message (1)							
	0	0	0	0	0	0	0	1
byte 3	Type (0xce) – unsigned integer							
	1	1	0	0	1	1	1	0
byte 4	Msgid (MSB)							
byte 5	Msgid (cont.)							
byte 6	Msgid (cont.)							
byte 7	Msgid (LSB)							
byte 8	Type – Nil							
	1	1	0	0	0	0	0	0
byte 9	Type – unsigned integer							
	0	0	0	0	0	0	1	1

Table 43 shows an example of an error occurred response message with a negative integer error code.

Table 43 – An example of an error occurred response message

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (4)			
	1	0	0	1	0	1	0	0
byte 2	Type – Response Message (1)							
	0	0	0	0	0	0	0	1
byte 3	Type (0xce) – unsigned integer							
	1	1	0	0	1	1	1	0
byte 4	Msgid (MSB)							
byte 5	Msgid (cont.)							
byte 6	Msgid (cont.)							
byte 7	Msgid (LSB)							
byte 8	Type – Negative integer							
	1	1	1	1	1	1	1	1
byte 9	Type – Nil							
	0	0	0	0	0	0	1	1

Table 44 shows an example of a success response message without result value. In this case, both the value of the error and results elements will be nil.

Table 44 – An example of a success response message without result

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (4)			
	1	0	0	1	0	1	0	0
byte 2	Type – Response Message (1)							
	0	0	0	0	0	0	0	1
byte 3	Type (0xce) – unsigned integer							
	1	1	0	0	1	1	1	0
byte 4	Msgid (MSB)							
byte 5	Msgid (cont.)							
byte 6	Msgid (cont.)							
byte 7	Msgid (LSB)							
byte 8	Type - Nil							
	1	1	0	0	0	0	0	0
byte 9	Type – Nil							
	1	1	0	0	0	0	0	0

5.4 Notify Message

A notify message shall consist of an array containing the three elements specified in Table 45.

Table 45 – The elements of a notify message

Element	Description
type	An integer that represents the type of message. This field shall contain “2” for a notify message.
method	A character string. This field shall contain the method name. The method name depends on the command definition which is beyond the scope of this document.
params	An object array or an object defined in Table 1. An object array shall be applied when compatibility with MessagePack RPC is required. This field shall contain the argument(s) of method. The applicable object depends on the command definition which is beyond the scope of this document.

Table 46 shows an example of a notify message. The method name is ‘Hello’, and the element params is an array consisting of an integer and a string.

Table 46 – An example of a notify message

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (3)			
	1	0	0	1	0	0	1	1
byte 2	Type – Notify Message (2)							
	0	0	0	0	0	0	1	0
byte 3	Type - string			String length				
	1	0	1	0	0	1	0	1
byte 4	‘H’							
byte 5	‘e’							
byte 6	‘l’							
byte 7	‘l’							
byte 8	‘o’							
byte 9	Type - array				Object number (2)			
	1	0	0	1	0	0	1	0
byte 10	0	0	0	0	0	0	1	1
byte 11	Type - string			String length (5)				
	1	0	1	0	0	1	0	1
byte 12	‘P’							
byte 13	‘a’							
byte 14	‘r’							
byte 15	‘a’							
byte 16	‘m’							

Table 47 shows an example of a notify message with no parameters. The method name is ‘Hello’, and the element params is an empty array.

Table 47 – An example of a notify message with no parameters

Bit	7	6	5	4	3	2	1	0
byte 1	Type - array				Object number (3)			
	1	0	0	1	0	0	1	1
byte 2	Type – Notify Message (2)							
	0	0	0	0	0	0	1	0
byte 3	Type - string			String length				
	1	0	1	0	0	1	0	1
byte 4	‘H’							
byte 5	‘e’							
byte 6	‘l’							
byte 7	‘l’							
byte 8	‘o’							
byte 9	Type - array				Object number (0)			
	1	0	0	1	0	0	0	0

6 Message Sequences

6.1 General

This section describes message sequences about a remote procedure call and notification using three kinds of message formats defined in Section 5.

6.2 Remote procedure call

Figure 1 shows a sequence on a remote procedure call. When the receiver receives a request message, it shall send a corresponding response message with same msgid to the sender.

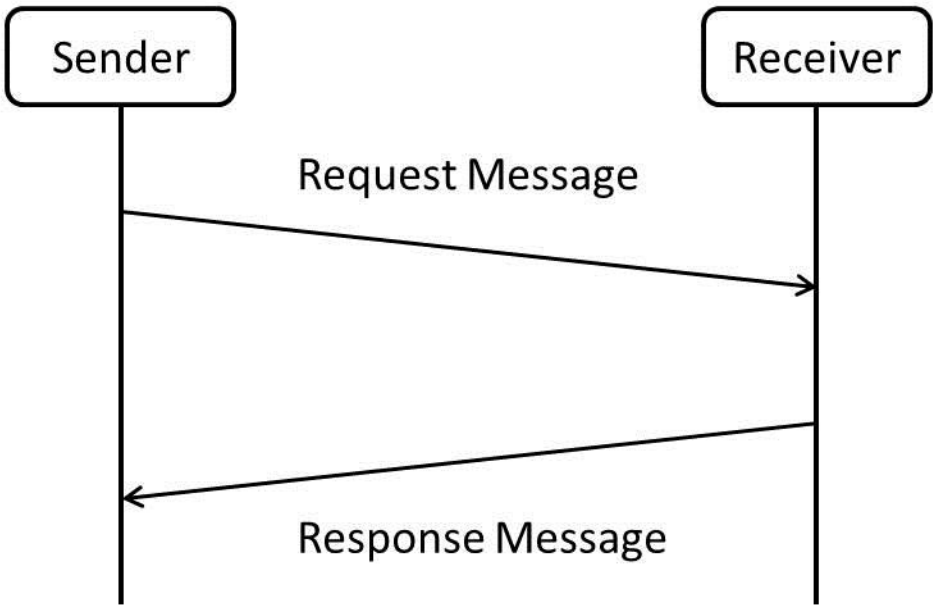

Figure 1 – Sequence on a remote procedure call

6.2.1 Order of the response messages

The receiver does not need to send response messages in the order of received request messages.

6.3 Notification

Figure 2 shows a sequence on a notification. A notify message is used for a notification. A notify message is a request message without a corresponding response message. Notify messages can be sent without any distinction between a client and a server, which realizes efficient communication between a controller and a device.

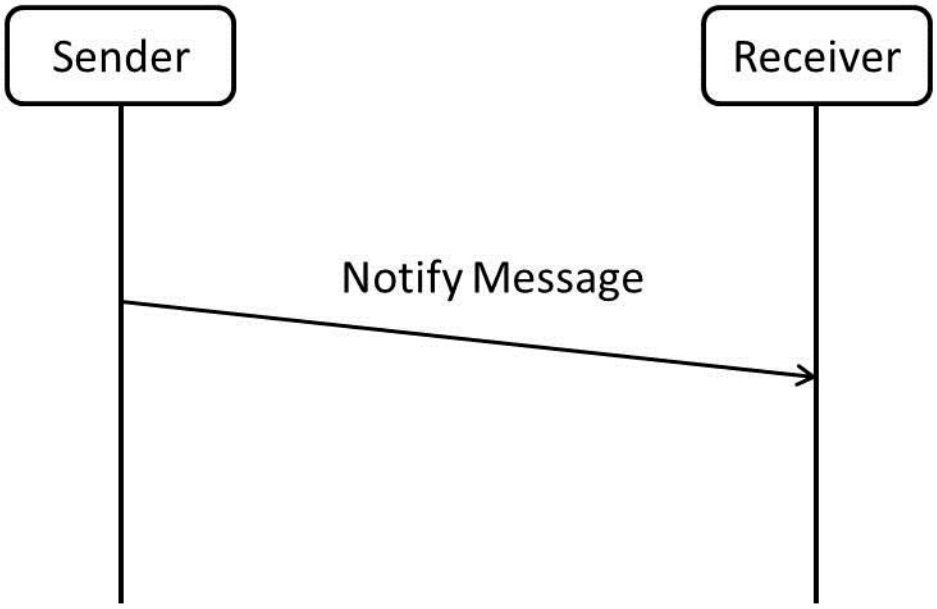

Figure 2 – Sequence on a notification

7 Transport Protocols

7.1 General

The following transport protocols can be used for transporting a remote procedure call and a notification described in Section 6.

7.2 WebSocket

A remote procedure call and a notification can be transported over a WebSocket connection defined in IETF RFC 6455. If user applications are Web applications, WebSocket or WebSocket Security shall be used as a transport protocol. If user applications also are native applications, WebSocket or WebSocket Security may be used as a transport protocol.

7.3 TLS

A remote procedure call and a notification can be transported over a TLS connection defined in IETF RFC 5246. If user applications are native applications, TLS may be used as a transport protocol.

7.4 TCP

A remote procedure call and a notification can be transported over a TCP connection. TCP may be used if WebSocket and TLS cannot be used as a transport protocol for some reasons, security is ensured in another way, or security is not required..

8 Security

8.1 General

The following security capabilities are supported by transport protocols described in Section 7.

8.2 User Authentication

The WebSocket supports a basic authentication and a digest authentication as defined in IETF RFC 2617. If an application requires a user authentication, WebSocket or WebSocket Security should be used as a transport protocol. A basic authentication should not be used if the communication path is not encrypted.

8.3 Client Authentication

The WebSocket Security and TLS support PKI authentication as a client authentication. If an application requires a client authentication, WebSocket Security or TLS should be used as a transport protocol.

8.4 Encryption of Communication Path

WebSocket Security or TLS should be used as a transport protocol if encryption of the communication path is required.

Annex A Reference Implementation (Informative)

A ‘linear-rpc’ source code reference implementation of this specification is available as open source at github.com, see below.

https://github.com/linear-rpc/