| Document Title | SOME/IP Protocol Specification |
|---|---|
| **Document Category** | Protocol specification (PRS) |
| **Document UID** | 696 |
| **Document Status** | published |
| **Part of AUTOSAR Standard** | Foundation |
| **Part of Standard Release** | R26-11 |

| Document Change History | | | |
|---|---|---|---|
| **Date** | **Release** | **Changed by** | **Description** |
| 2026-11-26 | R26-11 | AUTOSAR Release Management | • Added requirements on allowed characters in strings and padding of fixed-length strings<br>• Clarified server response addressing and concurrent requests handling<br>• Clarified handling of fixed-length strings exceeding the defined size and with reduced size<br>• Clarified Client ID uniqueness within a machine<br>• Removed requirement on compatibility of changes of the payload format<br>• Editorial Changes and bug fixes |
| 2025-11-27 | R25-11 | AUTOSAR Release Management | • Added Configuration parameters for (de-)serialization and referenced them in existing requirements<br>• Updated error handling flowchart, corresponding requirements and return codes table<br>• Reverted the handling of receiving less data than expected and substitution of missing data with default values during deserialization<br>• Clarified the requirements of deserialization with duplicate members and invalid wire type<br>• Editorial Changes and bug fixes |

▽

△

| | | | |
|---|---|---|---|
| 2024-11-27 | R24-11 | AUTOSAR Release Management | • Protocol updates for interoperability issues between AP and CP regarding SOME/IP Error responses<br>• Added configurable timer for SOME/IP-TP reception timeout time<br>• Editorial Changes |
| 2023-11-23 | R23-11 | AUTOSAR Release Management | • Clarifications in Payload Compatibility Rules table<br>• Changed [PRS_SOMEIP_00163] to allow sharing of local endpoint between different required service instances of the same service<br>• Made length field required in dynamic arrays<br>• Editorial Changes |
| 2022-11-24 | R22-11 | AUTOSAR Release Management | • Clarification of SOME/IP-TP segmentation<br>• Removed implementsSOMEIPStringHandling<br>• [PRS_SOMEIP_00300] extended by uint64<br>• Corrected serialization with BOM in [PRS_SOMEIP_00374]<br>• Corrected multiple Bugs in [PRS_SOMEIP_00043], [PRS_SOMEIP_00739], [PRS_SOMEIP_00043], [PRS_SOMEIP_00241], [PRS_SOMEIP_00101], [PRS_SOMEIP_00942] and [PRS_SOMEIP_00922]<br>• Added [PRS_SOMEIP_00245] to correct mismatch in size of Method-ID<br>• Editorial Changes |

▽

△

| | | | |
|---|---|---|---|
| 2021-11-25 | R21-11 | AUTOSAR Release Management | • Added Restriction of Server Connection from Clients<br>• Added Restriction of Client Connection to Server<br>• Clarification on String Handling in SOME/IP<br>• SOME/IP Header shall be encoded in network byte order<br>• Editorial Changes |
| 2020-11-30 | R20-11 | AUTOSAR Release Management | • Removed Draft Status from TLV Requirements<br>• Fixed discrepancies between SWS and PRS<br>• Clarified usage of length field<br>• Restricted alignment of variable length arrays to 8, 16, 32, 64, 128 or 256 Bits<br>• Editorial Changes |
| 2019-11-28 | R19-11 | AUTOSAR Release Management | • Added<br>– Support for unit64 / sint64<br>– Error-Codes for E2E-Protection<br>• Clarify<br>– Serialization of fixed length array data<br>– Support for Data Accumulation feature<br>– Contradicting requirements<br>• Introduce implementsLegacyStringSerialization tag (as successor of implementsSOMEIPStringHandling)<br>• Editorial Changes<br>• Changed Document Status from Final to published |
| 2019-03-29 | 1.5.1 | AUTOSAR Release Management | • No content changes |
| 2018-10-31 | 1.5.0 | AUTOSAR Release Management | • Backward-incompatibility statement removed<br>• Some statements improved |

▽

△

| | | | |
|---|---|---|---|
| 2018-03-29 | 1.4.0 | AUTOSAR Release Management | • Improved traceability |
| 2017-12-08 | 1.3.0 | AUTOSAR Release Management | • No content changes |
| 2017-10-27 | 1.2.0 | AUTOSAR Release Management | • Editorial changes |
| 2017-03-31 | 1.1.0 | AUTOSAR Release Management | • Serialization of Structured Datatypes and Arguments with Identifier and optional members |
| 2016-11-30 | 1.0.0 | AUTOSAR Release Management | • Initial Release |

**Disclaimer**

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

# Table of Contents

1 Introduction and overview 9
1.1 Protocol purpose and objectives . . . 9
1.2 Applicability of the protocol . . . 9
1.2.1 Constraints and assumptions . . . 9
1.2.2 Limitations . . . 10
1.3 Dependencies . . . 10
1.4 Document Structure . . . 10
2 Use Cases 11
3 Protocol Requirements 12
3.1 Requirements Traceability . . . 12
4 Acronyms and Abbreviations 13
5 Protocol specification 14
5.1 Specification of SOME/IP Message Format (Serialization) . . . 14
5.1.1 Header . . . 14
5.1.1.1 Message ID [32 Bit] . . . 16
5.1.1.2 Method ID [16 Bit] . . . 16
5.1.1.3 Length [32 Bit] . . . 16
5.1.1.4 Request ID [32 Bit] . . . 17
5.1.1.5 Protocol Version [8 Bit] . . . 19
5.1.1.6 Interface Version [8 Bit] . . . 19
5.1.1.7 Message Type [8 Bit] . . . 20
5.1.1.8 Return Code [8 Bit] . . . 20
5.1.1.9 Payload [variable size] . . . 21
5.1.2 Endianness . . . 21
5.1.3 Serialization of Data Structures . . . 22
5.1.3.1 Basic Datatypes . . . 24
5.1.3.2 Structured Datatypes (structs) . . . 25
5.1.3.3 Structured Datatypes and Arguments with Identifier and optional members (’TLV’) . . . 25
5.1.3.4 Strings . . . 33
5.1.3.5 Arrays . . . 35
5.1.3.6 Enumeration . . . 39
5.1.3.7 Bitfield . . . 39
5.1.3.8 Union / Variant . . . 39
5.1.4 De-serialization of Data Structures . . . 41
5.1.4.1 Structured DataTypes (structs) . . . 42
5.1.4.2 Structured Datatypes and Arguments with Identifier and optional members (’TLV’) . . . 42
5.1.4.3 Strings . . . 43

5.1.4.4 Arrays . . . 44
5.1.4.5 Enumeration . . . 45
5.1.4.6 Bitfield . . . 45
5.1.4.7 Union / Variant . . . 45
5.2 Specification of SOME/IP Protocol . . . 45
5.2.1 Transport Protocol Bindings . . . 45
5.2.1.1 UDP Binding . . . 46
5.2.1.2 TCP Binding . . . 47
5.2.1.3 Multiple Service-Instances . . . 49
5.2.1.4 Transporting large SOME/IP messages of UDP (SOME/IP-TP) 50
5.2.2 Request/Response Communication . . . 56
5.2.3 Fire&Forget Communication . . . 57
5.2.4 Notification Events . . . 58
5.2.4.1 Strategy for sending notifications . . . 59
5.2.5 Fields . . . 59
5.2.6 Error Handling . . . 60
5.2.6.1 Return Code . . . 61
5.2.6.2 Error Message . . . 62
5.2.6.3 Error Processing Overview . . . 62
5.3 Compatibility Rules for Interface Version . . . 66
6 Configuration Parameters 68
7 Protocol usage and guidelines 71
7.1 Choosing the transport protocol . . . 71
7.2 Security Considerations for SOME/IP . . . 71
7.3 Compatibility of changes of the payload format . . . 72
A Change history of AUTOSAR traceable items 78
A.1 Traceable item history of this document according to AUTOSAR Release R26-11 . . . 78
A.1.1 Added Specification Items in R26-11 . . . 78
A.1.2 Changed Specification Items in R26-11 . . . 78
A.1.3 Deleted Specification Items in R26-11 . . . 79
A.1.4 Added Constraints in R26-11 . . . 79
A.1.5 Changed Constraints in R26-11 . . . 79
A.1.6 Deleted Constraints in R26-11 . . . 79
A.2 Traceable item history of this document according to AUTOSAR Release R25-11 . . . 80
A.2.1 Added Specification Items in R25-11 . . . 80
A.2.2 Changed Specification Items in R25-11 . . . 80
A.2.3 Deleted Specification Items in R25-11 . . . 81
A.2.4 Added Constraints in R25-11 . . . 82
A.2.5 Changed Constraints in R25-11 . . . 82
A.2.6 Deleted Constraints in R25-11 . . . 82

A.3 Traceable item history of this document according to AUTOSAR Release R24-11 . . . 82
A.3.1 Added Specification Items in R24-11 . . . 82
A.3.2 Changed Specification Items in R24-11 . . . 83
A.3.3 Deleted Specification Items in R24-11 . . . 83
A.4 Traceable item history of this document according to AUTOSAR Release R23-11 . . . 84
A.4.1 Added Specification Items in R23-11 . . . 84
A.4.2 Changed Specification Items in R23-11 . . . 84
A.4.3 Deleted Specification Items in R23-11 . . . 84
A.5 Traceable item history of this document according to AUTOSAR Release R22-11 . . . 85
A.5.1 Added Specification Items in R22-11 . . . 85
A.5.2 Changed Specification Items in R22-11 . . . 85
A.5.3 Deleted Specification Items in R22-11 . . . 86

# 1 Introduction and overview

This protocol specification specifies the format, message sequences and semantics of the AUTOSAR Protocol **"Scalable service-Oriented MiddlewarE over IP (SOME/IP)"**.

SOME/IP is an automotive/embedded communication protocol which supports remote procedure calls, event notifications and the underlying serialization/wire format. The only valid abbreviation is SOME/IP. Other abbreviations (e.g. Some/IP) are wrong and shall not be used.

## 1.1 Protocol purpose and objectives

SOME/IP was created as existing solutions were not compatible to automotive use cases as well as incompatible to the AUTOSAR Classic Platform architecture. The major design goals were as follows:

- Fulfills the hard requirements regarding resource consumption of embedded ECUs
- Is compatible through as many use-cases and communication partners as possible
- Compatible with AUTOSAR at least on the wire-format level; i.e., SOME/IP was introduced with minimal architecture changes in the AUTOSAR Classic Platform.
- Provides the features required by automotive use-cases
- Is scalable from tiny to large platforms

## 1.2 Applicability of the protocol

SOME/IP shall be implemented on different operating system (i.e., AUTOSAR, GENIVI/COVESA, and OSEK) and even embedded devices without operating system. SOME/IP shall be used for inter-ECU Client/Server Serialization. An implementation of SOME/IP allows AUTOSAR to parse the RPC PDUs and transport the parameters to the application.

### 1.2.1 Constraints and assumptions

The "Support for serialization of extensible data structs" has been introduced - which SOME/IP serializers based on AUTOSAR Foundation Standard 1.0.0 (AUTOSAR Classic Standard 4.3.0) as well as most non-AUTOSAR implementations cannot process. To indicate this interoperability issue [PRS_SOMEIP_00220] requires to increase the major interface version of the respective serialized data.

The support for "complementary default value" during reception of less data than expected is no longer supported by AUTOSAR.

### 1.2.2 Limitations

**[constr_00001] Reordering limitation for a SOME/IP message segment** ⌈Reordering of out-of-order segments of a SOME/IP message is not supported.⌋

## 1.3 Dependencies

There are no dependencies to AUTOSAR SWS modules.

## 1.4 Document Structure

The SOME/IP PRS will describe the following two aspects of SOME/IP.

**Specification of SOME/IP on wire-format (Serialization)**

- Structure of Header Format
- How the different data types are serialized as per SOME/IP

**Specification of Protocol for Event and RPC-based communication**

- Transport Protocol
- Rules that govern the RPC for SOME/IP

In addition to this document, the PRS SOME/IP-SD [1] describes the Service Discovery and other functionalities of SOME/IP.

# 2 Use Cases

| *ID* | *Name* | *Description* |
|---|---|---|
| **UC_SOMEIP_00001** | Remote Procedure Call: Request/Response Methods | A Client application sends request to the server application and expects back a response from Server application. A Server application sends back response to the Client application on receiving a request. |
| **UC_SOMEIP_00002** | Remote Procedure Call: Fire and Forget Methods | A Client application sends request to the server application and does not expect any response. |
| **UC_SOMEIP_00003** | Event based communication | A Server application sends events to relevant client applications that are determined via SOME/IP-SD. |
| **UC_SOMEIP_00004** | Field based communication | Applications communicate using fields that can have an optional notifier (event to be sent cyclically or on-change), an optional setter (a request/response method to update the field), and an optional getter (a request/response method to read out the current value of the field). |
| **UC_SOMEIP_00005** | Communication using various transport protocols | Applications communicate over UDP or TCP as an underlying transport protocol. |
| **UC_SOMEIP_00006** | Communication based on Publish and Subscribe model | Server applications dynamically offer and publish data to the required client applications in the network. Client applications dynamically subscribe and receive data from the required server applications in the network. |
| **UC_SOMEIP_00007** | Message Processing and Error Handling | Applications process the messages in a structured way and perform the error handling using specific Return Codes and Error Messages. |
| **UC_SOMEIP_00008** | Different platform support | Applications based on different implementation platforms and Operating Systems communicate with each other fulfilling the hard requirements of resource consumption. |

# 3 Protocol Requirements

## 3.1 Requirements Traceability

| Requirement | Description | Satisfied by |
|---|---|---|
| No upstream requirements specified. | | |

**Table 3.1: Requirements Tracing**

# 4 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the SOME/IP specification that are not included in the [2, AUTOSAR glossary].

| Abbreviation / Acronym: | Description: |
|---|---|
| Byte Order Mark | The byte order mark (BOM) is a Unicode character, U+FEFF byte order mark (BOM), whose appearance as a magic number at the start of a text stream is used to indicate the used encoding. |
| Method | A method, procedure, function, or subroutine that is called/invoked. |
| Parameters | input, output, or input/output arguments of a method or an event |
| Remote Procedure Call (RPC) | A method call from one ECU to another that is transmitted using messages |
| Request | a message of the client to the server invoking a method |
| Response | a message of the server to the client transporting results of a method invocation |
| Request/Response communication | a RPC that consists of request and response |
| Event | A uni-directional data transmission that is only invoked on changes or cyclically and is sent from the producer of data to the consumers. |
| Field | A field does represent a status and thus has an valid value at all times on which getter, setter and notifier act upon. |
| Notification Event | An event message of the notifier of a field. |
| Getter | A Request/Response call that allows read access to a field. |
| Setter | A Request/Response call that allows write access to a field. |
| Notifier | Sends out event message with the fields value on change, on epsilon change, or cyclically based on configuration. |
| Service | A logical combination of zero or more methods, zero or more events, and zero or more fields. |
| Service Interface | the formal specification of the service including its methods, events, and fields |
| Eventgroup | A logical grouping of events and notification events of fields inside a service in order to allow subscription |
| Service Instance | Implementation of a service, which can exist more than once in the vehicle and more than once on an ECU |
| Server | The ECU offering a service instance shall be called server in the context of this service instance. |
| Client | The ECU using the service instance of a server shall be called client in the context of this service instance. |
| Fire and Forget | Requests without response message are called fire&forget. |
| User Datagram Protocol | A standard network protocol using a simple connectionless communication model. |
| Union | A data structure that dynamically assumes different data types. |
| non-extensible (standard) struct | A struct which is serialized without tags. At most, new members can be added in a compatible way at the end of the struct and optional members are not possible. |
| extensible struct | A struct which is serialized with tags. New members can be added in a compatible way at arbitrary positions and optional members are possible. |
| TLV | Tag Length Value |

**Table 4.1: Acronyms and Abbreviations**

# 5 Protocol specification

SOME/IP provides service oriented communication over a network. It is based on service definitions that list the functionality that the service provides. A service can consist of combinations of zero or multiple events, methods and fields.

Events provide data that are sent cyclically or on change from the provider to the subscriber.

Methods provide the possibility to the subscriber to issue remote procedure calls which are executed on provider side.

Fields are combinations of one or more of the following three

- a notifier which sends data from the provider to the subscribers on change, on epsilon change, or cyclically based on configuration.
- a getter which can be called by the subscriber to explicitly query the provider for the value
- a setter which can be called by the subscriber when it wants to change the value on provider side

The major difference between the notifier of a field and an event is that events are only sent on change, the notifier of a field additionally sends the data directly after subscription.

## 5.1 Specification of SOME/IP Message Format (Serialization)

Serialization describes the way data is represented in protocol data units (PDUs) as payload of either UDP or TCP messages, transported over an IP-based automotive in-vehicle network.

### 5.1.1 Header

**[PRS_SOMEIP_00030]** ⌈The structure of header layout shall consist of

- Message ID (Service ID/Method ID) [32 Bits]
- Length [32 Bits]
- Request ID (Client ID/Session ID) [32 Bits]
- Protocol Version [8Bits]
- Interface Version [8 Bits]
- Message Type [8 Bits]
- Return Code [8 Bits]

⌋

[PRS_SOMEIP_00030] is shown in Table 5.1.

| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | | | |
|---|---|---|---|
| Message ID (Service ID / Method ID) [32 Bit] | | | |
| Length [32 Bit] | | | |
| Request ID (Client ID / Session ID) [32 Bit] | | | |
| Protocol Version [8 Bit] | Interface version [8 Bit] | Message Type [8 Bit] | Return Code [8 Bit] |
| Payload [variable size] | | | |

**Table 5.1: SOME/IP Header Format**

**[PRS_SOMEIP_00941]** ⌈In case of E2E communication protection being applied, the E2E header is placed after Return Code, depending on the chosen Offset value for the E2E header. The default Offset value is 64 bit, which puts the E2E header exactly between Return Code and Payload.⌋

[PRS_SOMEIP_00941] is shown in the Table 5.2.

| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | | | |
|---|---|---|---|
| Message ID (Service ID / Method ID) [32 Bit] | | | |
| Length [32 Bit] | | | |
| Request ID (Client ID / Session ID) [32 Bit] | | | |
| Protocol Version [8 Bit] | Interface version [8 Bit] | Message Type [8 Bit] | Return Code [8 Bit] |
| E2E Header (variable size/ Dependent on selected E2E profile] | | | |
| Payload [variable size] | | | |

**Table 5.2: SOME/IP Header and E2E header Format**

**[PRS_SOMEIP_00031]** ⌈For interoperability reasons the header layout shall be identical for all implementations of SOME/IP. The fields are presented in transmission order i.e. the fields on the top left are transmitted first.⌋

**Note:** For the sake of proper processing (see e.g. Chapter 5.2.6.3 Error Processing Overview) involving different protocol versions, the location, the length and the encoding of the following SOME/IP header fields will remain the same in all upcoming SOME/IP protocol versions:

- Message ID (consisting of Service ID and Event/Method ID)
- Length
- Protocol Version

#### 5.1.1.1 Message ID [32 Bit]

**[PRS_SOMEIP_00034]** ⌈The Message ID shall be a 32 Bit identifier that is used to identify

- the RPC call to a method of an application
- or to identify an event.

⌋

**Note:** The assignment of the Message ID is up to the user / system designer. However, the Message ID is assumed be unique for the whole system (i.e. the vehicle).

#### 5.1.1.2 Method ID [16 Bit]

**[PRS_SOMEIP_00245]** ⌈The Message ID header field shall be structured into a 16 Bit Service ID header field (to distinguish up to $2^{16}$ services) and a 16 bit Method ID header field to distinguish up to $2^{16}$ service elements (namely methods and/or events). This structuring of the Message ID header field is illustrated as shown in [PRS_SOMEIP_00755].⌋

**Note:** It is common practice and recommended to split the ID space of the Method ID between Methods and Events/Notifications. Methods would be in the range `0x0000-0x7FFF` (first bit of Method-ID is 0) and Events/Notifications would use the range `0x8000-0x8FFF` (first bit of the Method-ID is 1).

**[PRS_SOMEIP_00755] Message ID header field** ⌈

| Service ID [16 Bit] | Method ID [16 Bit] |
|---|---|

⌋

Eventgroup is a logical grouping of events and notification events of fields inside a service in order to allow subscription.

**[PRS_SOMEIP_00365]** ⌈A SOME/IP Eventgroup shall at least contain one event.⌋

**[PRS_SOMEIP_00366]** ⌈Events as well as field notifiers shall be mapped to at least one SOME/IP Eventgroup.⌋

#### 5.1.1.3 Length [32 Bit]

**[PRS_SOMEIP_00042]** ⌈Length field shall contain the length in Byte starting from Request ID/Client ID until the end of the SOME/IP message.⌋

#### 5.1.1.4 Request ID [32 Bit]

The Request ID allows a server and client to differentiate multiple parallel uses of the same method, getter or setter.

**[PRS_SOMEIP_00043]** ⌈The Request ID shall be unique for a request-response pair to differentiate between multiple calls of the same method.⌋

**[PRS_SOMEIP_00704]** ⌈When generating a response message, the provider shall copy the Request ID from the request to the response message.⌋

**Note:**
This allows the client to map a response to the issued request even with more than one request outstanding.

**[PRS_SOMEIP_00044]** ⌈Request IDs must not be reused until the response has arrived or is not expected to arrive anymore (timeout).⌋

**Structure of the Request ID**

**[PRS_SOMEIP_00046] Request ID** ⌈

| Client ID [16 Bits] | Session ID [16 Bits] |
|---|---|

⌋

**Note:**
This means that the implementer of an ECU can define the Client-IDs as required by his implementation and the provider does not need to know this layout or definitions because he just copies the complete Request-ID in the response.

**[PRS_SOMEIP_00702] Client ID uniqueness within a machine** ⌈Each SOME/IP client on one ECU/Machine shall use one unique Client ID. This applies to all clients, independent of the used services or service instances.⌋

**Rationale**:
Unique Client IDs are required to correctly routing method responses to the correct client (IP address and UDP/TCP port number alone are insufficient). Uniqueness across different UDP/TCP port numbers is required because at the time the Client ID is assigned the UDP/TCP port number may not yet be known.

**[PRS_SOMEIP_00949] Server Response Addressing** ⌈The server shall respond using the addressing information (IP address and UDP/TCP port number) from the initiating request message and shall forward the Request ID unchanged.⌋

**Rationale**:
The server does not interpret the Request ID, it is only forwarded by the server. The

forwarded Request ID in this case enables the client to distinguish multiple concurrent requests originating from the same socket endpoint, and therefore the Request ID must be unique for each parallel request.

**[PRS_SOMEIP_00950] Concurrent Requests Handling** ⌈The server shall be able to handle multiple concurrent requests from the same addressing information (IP address and UDP/TCP port number).⌋

**[PRS_SOMEIP_00703]** ⌈The Session ID is a unique identifier that allows to distinguish sequential messages or requests originating from the same sender from each other.⌋

**[PRS_SOMEIP_00532]** ⌈The Client ID shall support a configuration that enables uniqueness across the overall vehicle by having dedicated distinct ranges, a configurable prefix, or fixed value per ECU/Machine.⌋

**Note**:
The assignment of the Client IDs is up to the user / system designer, however [PRS_SOMEIP_00702] always needs to be respected.
The additional option of vehicle wide uniquenes of Client IDs as described in [PRS_SOMEIP_00532] can be realized with different approaches, e.g. by using the most significant byte of Client ID being the diagnostics address, a configured Client ID for a given application / SWC, or defining a dedicated Client ID Range for each ECU.

For example:

| Client ID Prefix [8 Bits] | Client ID [8 Bits] | Session ID [16 Bits] |
|---|---|---|

**Table 5.3: Example of Client ID**

**[PRS_SOMEIP_00932]** ⌈In case Session Handling is not active, the Session ID shall be set to `0x00`.⌋

**[PRS_SOMEIP_00933]** ⌈In case Session Handling is active, the Session ID shall be set to a value within the range `[0x1, 0xFFFF]`.⌋

**[PRS_SOMEIP_00934]** ⌈In case Session Handling is active, the Session ID shall be incremented according to the respective use case (detailed information about dedicated use cases is contained in separate specification items (e.g., [PRS_SOMEIP_00533]).⌋

**[PRS_SOMEIP_00533]** ⌈Request/Response methods shall use session handling with Session IDs. Session ID should be incremented after each call.⌋

**[PRS_SOMEIP_00521]** ⌈When the Session ID reaches 0xFFFF, it shall wrap around and start again with `0x01`⌋

**[PRS_SOMEIP_00739]** ⌈For request/response methods, a client has to ignore a response if the Session ID of the response does not match the Session ID of the request⌋

**[PRS_SOMEIP_00935]** ⌈For notification messages, a receiver shall ignore the Session ID in case Session Handling is not active.⌋

**[PRS_SOMEIP_00936]** ⌈For notification messages, a receiver shall treat the Session ID according to the respective use case (detailed information about dedicated use cases is contained in separate specification items [e.g., [PRS_SOMEIP_00741]] in case Session Handling is active).⌋

**[PRS_SOMEIP_00939] Session Handling for `REQUEST_NO_RETURN` Messages** ⌈Session Handling shall not be used for messages of type `REQUEST_NO_RETURN`, i.e, the Session ID part of the Request ID field of the SOME/IP header shall be 0x00 for messages of type `REQUEST_NO_RETURN` if segmentation (SOME/IP-TP) is not required as per [PRS_SOMEIP_00720].⌋

#### 5.1.1.5 Protocol Version [8 Bit]

The Protocol Version identifies the used SOME/IP Header format (not including the Payload format).

**[PRS_SOMEIP_00052]** ⌈Protocol Version shall be an 8 Bit field containing the SOME/IP protocol version.⌋

**[PRS_SOMEIP_00050]** ⌈The Protocol Version shall be increased, for all incompatible changes in the SOME/IP header. A change is incompatible if a receiver that is based on an older Protocol Version would not discard the message and process it incorrectly.

**Note:**
The Protocol Version will not be increased for changes that only affect the payload format.⌋

**Note:**
Message processing and error handling is defined in chapter 4.2.6.3 (error processing overview)

**[PRS_SOMEIP_00051]** ⌈The Protocol Version shall be 1.⌋

#### 5.1.1.6 Interface Version [8 Bit]

**[PRS_SOMEIP_00053]** ⌈Interface Version shall be an 8 Bit field that contains the Major Version of the Service Interface.⌋

#### 5.1.1.7 Message Type [8 Bit]

**[PRS_SOMEIP_00055] Message Type field** ⌈

| Number | Value | Description |
|---|---|---|
| 0x00 | REQUEST | A request expecting a response (even void) |
| 0x01 | REQUEST_NO_RETURN | A fire&forget request |
| 0x02 | NOTIFICATION | A request of a notification/event callback expecting no response |
| 0x20 | TP_REQUEST | A TP request expecting a response (even void) |
| 0x21 | TP_REQUEST_NO_RETURN | A TP fire&forget request |
| 0x22 | TP_NOTIFICATION | A TP request of a notification/event callback expecting no response |
| 0x40-0x42 | | Reserved by ISO 17215 |
| 0x80 | RESPONSE | The response message |
| 0x81 | ERROR | The response containing an error |
| 0xa0 | TP_RESPONSE | The TP response message |
| 0xa1 | TP_ERROR | The TP response containing an error |
| 0xc0-0xc1 | | Reserved by ISO 17215 |
| All other values | | Reserved for future assignment by AUTOSAR |

⌋

**[PRS_SOMEIP_00701]** ⌈Regular request (message type 0x00) shall be answered by a response (message type 0x80) with return code 0x00 (E_OK) when no error occurred. If error occurs, then either a response message (message type 0x80) or an error message (message type 0x81) with return code not equal to 0x00 (E_OK) shall be sent.⌋

**Note:** Refer [PRS_SOMEIP_00901] and [PRS_SOMEIP_00903], that describe when messages with message type response and message type error can be sent.

It is also possible to send a request that does not have a response message (message type 0x01). For updating values through notification a callback interface exists (message type 0x02).

**[PRS_SOMEIP_00367]** ⌈The 3rd highest bit of the Message Type (=0x20) shall be called TP-Flag and shall be set to 1 to signal that the current SOME/IP message is a segment. The other bits of the Message Type are set as specified in this Section.⌋

**Note:**
Segments of the Message Type Request (0x00) have the Message Type (0x20), segments of the Message Type Response (0x80) have the Message Type (0xa0), and so on. For details see (Chapter 5.2.1.4)

#### 5.1.1.8 Return Code [8 Bit]

**[PRS_SOMEIP_00058]** ⌈The Return Code shall be used to signal whether a request was successfully processed. For simplification of the header layout, every message

transports the field Return Code. The allowed Return Codes for specific message types are shown [PRS_SOMEIP_00757].⌋

**[PRS_SOMEIP_00757] Return Codes** ⌈

| Message Type | Allowed Return Codes |
|---|---|
| REQUEST | N/A set to 0x00 (E_OK) |
| REQUEST_NO_RETURN | N/A set to 0x00 (E_OK) |
| NOTIFICATION | N/A set to 0x00 (E_OK) |
| RESPONSE | See Return Codes in [PRS_SOMEIP_00191] |
| ERROR | See Return Codes in [PRS_SOMEIP_00191]. Shall not be 0x00 (E_OK). |

⌋

#### 5.1.1.9 Payload [variable size]

In the payload field the parameters are carried. The serialization of the parameters will be specified in the following section.

The size of the SOME/IP payload field depends on the transport protocol used.

**[PRS_SOMEIP_00382] SOME/IP payload size using UDP** ⌈With UDP the SOME/IP payload size should be between 0 and 1400 Bytes. Payload sizes greater than 1400 Bytes should be supported with TCP and segmentation of payload (see also [PRS_SOMEIP_00730]).⌋

**Note:**
The recommendation to 1400 Bytes improves change compatibility on the protocol stack (e.g. changing to IPv6 or adding security means).

Payload might consists of data elements for events or parameters for methods.

### 5.1.2 Endianness

**[PRS_SOMEIP_00368]** ⌈All SOME/IP Header Fields shall be encoded in network byte order (big endian).⌋

**[PRS_SOMEIP_00759]** ⌈The following fields in the payload of SOME/IP messages shall be encoded in network byte order (big endian):

- the optional length field of extensible structs ([PRS_SOMEIP_00079]),
- the TLV tag ([PRS_SOMEIP_00203]) and the length field ([PRS_SOMEIP_00221]) of structured datatypes and arguments with identifier and optional members,
- the optional length field for fixed length strings ([PRS_SOMEIP_00760]),

- the length field for dynamic length strings ([PRS_SOMEIP_00089]),
- the optional length field for extensible fixed length arrays ([PRS_SOMEIP_00944]),
- the length field of dynamic length arrays ([PRS_SOMEIP_00376]),
- the length field of unions ([PRS_SOMEIP_00126]),
- and the type field of unions ([PRS_SOMEIP_00129]).

⌋

**[PRS_SOMEIP_00369]** ⌈The byte order of the parameters inside the payload shall be determined by configuration parameter BYTE_ORDER in the allowed values according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter BYTE_ORDER, the default value defined in [PRS_SOMEIP_00004] shall be used as the byte order of the parameters. Exceptions are some of the fields in the payload of SOME/IP messages listed in [PRS_SOMEIP_00759], which shall be encoded in Network Byte Order regardless of the value of configuration parameter BYTE_ORDER.⌋

### 5.1.3 Serialization of Data Structures

The serialization is based on the parameter list defined by the interface specification. The interface specification defines the exact position of all data structures in the PDU and has to consider the memory alignment.

Alignment is used to align the beginning of data by inserting padding elements after the data in order to ensure that the aligned data starts at certain memory addresses.

There are processor architectures which can access data more efficiently (i.e. master) when they start at addresses which are multiples of a certain number (e.g multiples of 32 Bit).

**[PRS_SOMEIP_00611]** ⌈Alignment of data shall be realized by inserting padding elements after the variable size data.⌋

**Note:**
This applies for all hierachical levels.
Please note that the padding value is not defined.

**Example:** Structure with 5 Members
- **Member1**: UINT16
- **Member2**: One dimensional variableSize Array with uint8 elements
- **Member3**: UINT32
- **Member4**: UINT64
- **Member5**: One dimensional variableSize Array with uint8 elements

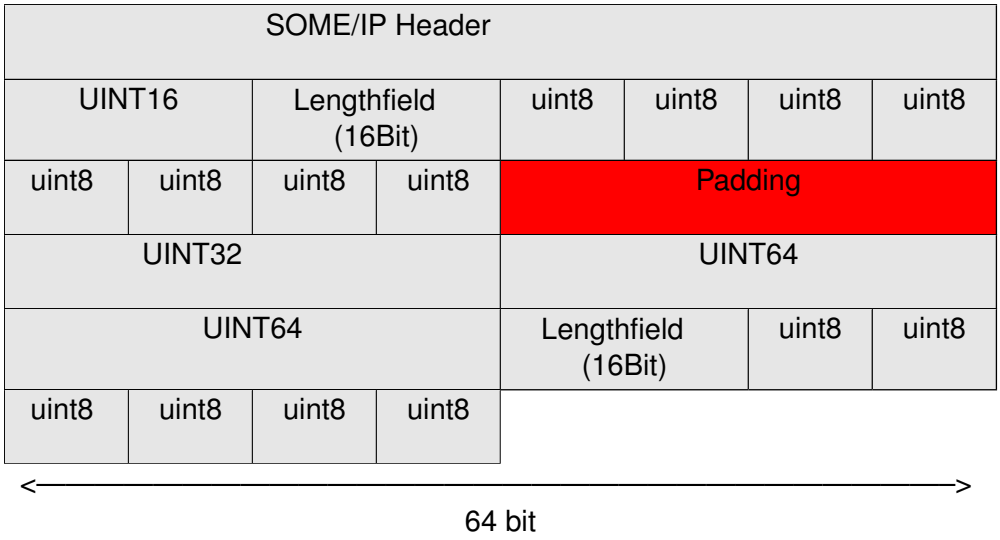


**Table 5.4: SOME/IP Padding Example 01**

**Example:** Structure with 5 Members
- **Member1**: UINT16
- **Member2**: One dimensional variableSize Array with uint8 elements
- **Member3**: UINT32
- **Member4**: UINT64
- **Member5**: One dimensional variableSize Array with uint8 elements

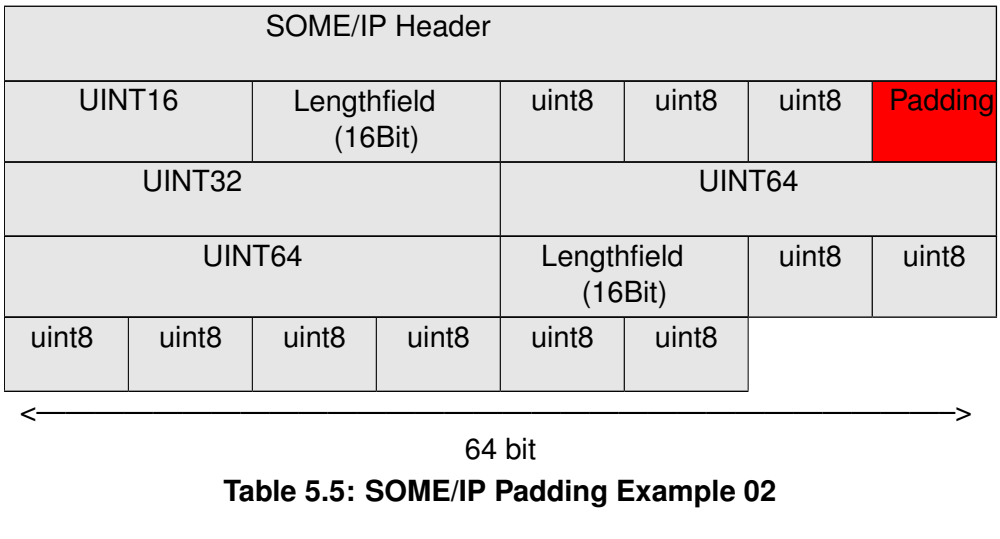


**Table 5.5: SOME/IP Padding Example 02**

**[PRS_SOMEIP_00569]** ⌈Alignment shall always be calculated from start of SOME/IP message.⌋

**[PRS_SOMEIP_00612]** ⌈There shall be no padding behind fixed length data elements to ensure alignment of the following data.⌋

**Note:**
If data behind fixed length data elements shall be padded, this has to be explicitly considered in the data type definition.

**[PRS_SOMEIP_00613]** ⌈The alignment for the data element immediately following a variable length data element (if it is not the last element in the serialized data stream) shall be determined by the configuration parameter ALIGNMENT in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter ALIGNMENT, the default value defined in [PRS_SOMEIP_00004] shall be used as alignment.⌋

#### 5.1.3.1 Basic Datatypes

**[PRS_SOMEIP_00065] Supported basic Data Types** ⌈

| Type | Description | Size [bit] | Remark |
|---|---|---|---|
| boolean | TRUE/FALSE value | 8 | FALSE (0), TRUE (1) |
| uint8 | unsigned Integer | 8 | |
| uint16 | unsigned Integer | 16 | |
| uint32 | unsigned Integer | 32 | |
| uint64 | unsigned Integer | 64 | |
| sint8 | signed Integer | 8 | |
| sint16 | signed Integer | 16 | |
| sint32 | signed Integer | 32 | |
| sint64 | signed Integer | 64 | |
| float32 | floating point number | 32 | IEEE 754 binary32 (Single Precision) |
| float64 | floating point number | 64 | IEEE 754 binary64 (Double Precision) |

⌋

The Byte Order is specified for each parameter by configuration.

**[PRS_SOMEIP_00615]** ⌈For the evaluation of a Boolean value only the lowest bit of the uint8 is interpreted and the rest is ignored.⌋

#### 5.1.3.2 Structured Datatypes (structs)

The serialization of a struct shall be close to the in-memory layout. This means, only the parameters shall be serialized sequentially into the buffer. Especially for structs it is important to consider the correct memory alignment.

Example:

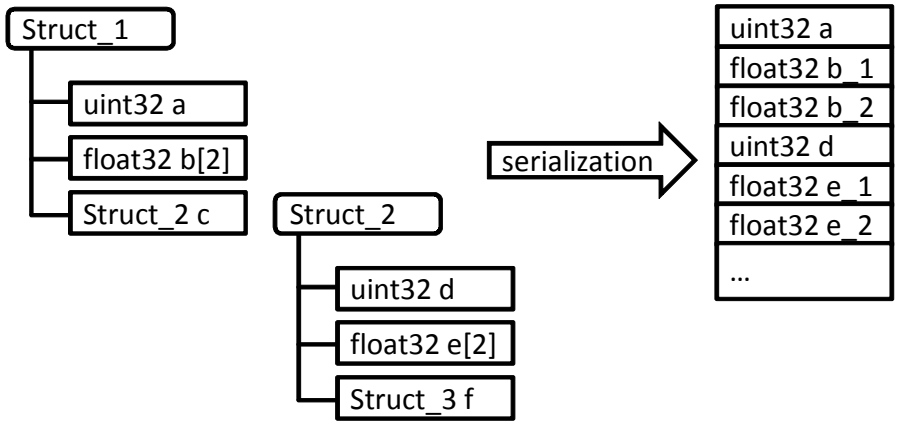


**Figure 5.1: Serialization of Structs**

**[PRS_SOMEIP_00077]** ⌈The SOME/IP implementation shall not automatically insert dummy/padding data.⌋

**[PRS_SOMEIP_00079]** ⌈A length field may be inserted in front of the Struct depending on the configuration parameter SIZE_OF_STRUCT_LENGTH_FIELD in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter SIZE_OF_STRUCT_LENGTH_FIELD, the default value defined in [PRS_SOMEIP_00004] shall be used for the length of the length field.⌋

**[PRS_SOMEIP_00370]** ⌈The length field of the struct shall describe the number of bytes this struct occupies for SOME/IP transport including potential length fields and potential padding (according to [PRS_SOMEIP_00611]) of the individual struct elements of the struct but excluding the struct's own length field.⌋

**[PRS_SOMEIP_00712]** ⌈The serialization of structs shall follow the depth-first-traversal of the structured data type.⌋

#### 5.1.3.3 Structured Datatypes and Arguments with Identifier and optional members ('TLV')

To achieve enhanced forward and backward compatibility, an additional Data ID can be added in front of struct members or method arguments. The receiver then can skip unknown members/arguments, i.e. where the Data ID is unknown. New members/arguments can be added at arbitrary positions when Data IDs are transferred in the serialized byte stream.

Moreover, the usage of Data IDs allows describing structs and methods with optional members/arguments. Whether a member/argument is optional or not, is defined in the data definition.

Whether an optional member/argument is actually present in the struct/method or not, must be determined during runtime. How this is realized depends on the used programming language or software platform (e.g. using a special available flag, using a special method, using pointers which might be null, ...).

**[PRS_SOMEIP_00201]** ⌈A Data ID shall be unique within the direct members of a struct or arguments of a method.⌋

**Note:**
Please note that a Data ID does not need to be unique across different structs or methods.

**Note:**
Please note that neither the AUTOSAR Methodology nor AUTOSAR CP RTE, nor AUTOSAR AP ara::com support the definition or usage of optional method arguments at the time being.

**[PRS_SOMEIP_00230]** ⌈A Data ID shall be defined either for all members of the same hierarchical level of a struct or for none of them.⌋

**[PRS_SOMEIP_00231]** ⌈A Data ID shall be defined either for all arguments of a method or for none of them.⌋

In addition to the Data ID, a wire type encodes the datatype of the following member. Data ID and wire type are encoded in a so-called tag.

**[PRS_SOMEIP_00202]** ⌈The length of a tag shall be two bytes.⌋

**[PRS_SOMEIP_00203]** ⌈The tag shall consist of

- reserved (Bit 7 of the first byte)
- wire type (Bit 6-4 of the first byte)
- Data ID (Bit 3-0 of the first byte and bit 7-0 of the second byte)

⌋

Refer to the Figure 5.2 for the layout of the tag. Bit 7 is the highest significant bit of a byte, bit 0 is the lowest significant bit of a byte.

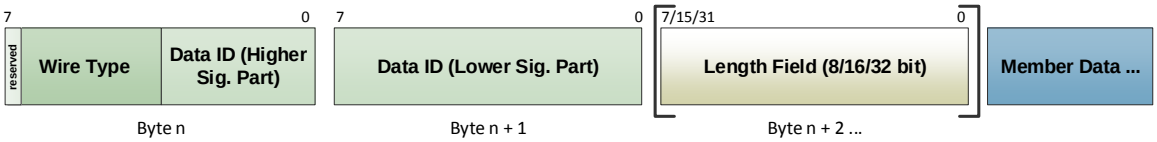


**Figure 5.2: Tag Layout**

**[PRS_SOMEIP_00204]** ⌈The lower significant part of the Data ID of the member shall be encoded in bits 7-0 of the second byte of the tag. The higher significant part of the Data ID of the member shall be encoded in bits 3-0 of the first byte.⌋

**Example:**
The Data ID of the member is 0x04F2. Then bits 3-0 of the first byte are set to 0x4. The second byte is set to 0xF2.

**[PRS_SOMEIP_00205] Wire Type and type of following data** ⌈

| Wire Type | Following Data |
|---|---|
| 0 | 8 Bit Data Base data type |
| 1 | 16 Bit Data Base data type |
| 2 | 32 Bit Data Base data type |
| 3 | 64 Bit Data Base data type |
| 4 | Complex Data Type: Array, Struct, String, Union with length field of static size (configured in data definition) |
| 5 | Complex Data Type: Array, Struct, String, Union with length field size 1 byte (ignore static definition) |
| 6 | Complex Data Type: Array, Struct, String, Union with length field size 2 byte (ignore static definition) |
| 7 | Complex Data Type: Array, Struct, String, Union with length field size 4 byte (ignore static definition) |

⌋

**Note:**
wire type 4 ensures the compatibility with the current approach where the size of length fields is statically configured. This approach has the drawback that changing the size of the length field during evolution of interfaces is always incompatible. Thus, wire types 5, 6 and 7 allow to encode the size of the used length field in the transferred byte stream. A serializer may use this, if the statically configured size of the length field is not sufficient to hold the current size of the data struct.

**[PRS_SOMEIP_00206]** ⌈On reception side, if the wire type is set to 5, 6 or 7, the size of the length field defined in the data definition shall be ignored and the size of the length field shall be selected according to the wire type.⌋

If a Data ID is configured for a member of a struct/argument of a method, a tag shall be inserted in the serialized byte stream.

**Note:**
regarding the existence of Data IDs, refer to [PRS_SOMEIP_00230] and [PRS_SOMEIP_00231].

**[PRS_SOMEIP_00003] Selection of to-be-used wire type and length of the Length field** ⌈On transmission side, the selection of the to-be-used wire type and the corresponding length of the length field shall depend on the configuration parameter IS_DYNAMIC_LENGTH_FIELD_SIZE. If the configuration parameter IS_DYNAMIC_LENGTH_FIELD_SIZE is set to TRUE, then a wire type of 5, 6 or 7 and a corresponding length for the length field (i.e., 1, 2, or 4 Bytes) shall be used depending on the actual size of the member data. If the configuration parameter IS_DYNAMIC_LENGTH_FIELD_SIZE is set to FALSE, then a wire type of 4 shall be used and the length of the length field shall be selected according to [PRS_SOMEIP_00001](SIZE_OF_ARRAY_LENGTH_FIELD),[PRS_SOMEIP_00002](SIZE_OF_STRING_LENGTH_FIELD),[PRS_SOMEIP_00079](IS_DYNAMIC_LENGTH_FIELD_SIZE) and [PRS_SOMEIP_00121](SIZE_OF_UNION_LENGTH_FIELD). If not configured explicitly via the configuration parameter IS_DYNAMIC_LENGTH_FIELD_SIZE, the default value defined in [PRS_SOMEIP_00004] shall be used for the selection of the to-be-used wire type and the corresponding length of the length field.⌋

**[PRS_SOMEIP_00212]** ⌈

If the datatype of the serialized member/argument is a basic datatype (wire types 0-3) and a Data ID is configured, the tag shall be inserted directly in front of the member/argument. No length field shall be inserted into the serialized stream.⌋

**[PRS_SOMEIP_00213]** ⌈If the datatype of the serialized member/argument is not a basic datatype (wire type 4-7) and a Data ID is configured, the tag shall be inserted in front of the length field.⌋

**[PRS_SOMEIP_00214]** ⌈If the datatype of the serialized member/argument is not a basic datatype and a Data ID is configured, a length field shall always be inserted in front of the member/argument.⌋

**Rationale:**
The length field is required for the de-serialization of known members/arguments and to skip unknown members/arguments during deserialization.

**[PRS_SOMEIP_00221]** ⌈The length field shall always contain the length up to the next tag of the struct.⌋

**[PRS_SOMEIP_00208]** ⌈If the members/arguments itself are of type struct, there shall be exactly one length field. The length field is added according to requirements [PRS_SOMEIP_00079] and [PRS_SOMEIP_00370].⌋

**[PRS_SOMEIP_00225]** ⌈If the members/arguments itself are of type dynamic length string, there shall be exactly one length field. The length field is added according to requirements [PRS_SOMEIP_00089], [PRS_SOMEIP_00090], [PRS_SOMEIP_00002] and [PRS_SOMEIP_00095].⌋

**[PRS_SOMEIP_00224]** ⌈If the members/arguments itself are of type fixed length string, there shall be exactly one length field corresponding to dynamic length strings.⌋

**Note:**
when serialized without tag, fixed length strings do not have a length field. For the serialization with tag, a length field is also required for fixed length strings in the same way as for dynamic length strings.

**[PRS_SOMEIP_00227]** ⌈If the members/arguments itself are of type dynamic length array, there shall be exactly one length field. The length field is added according to requirements [PRS_SOMEIP_00376], [PRS_SOMEIP_00001], [PRS_SOMEIP_00377] with a size of 8, 16 or 32 bit.⌋

**[PRS_SOMEIP_00226]** ⌈If the members/arguments itself are of type fixed length array, there shall be exactly one length field corresponding to dynamic length arrays.⌋

**[PRS_SOMEIP_00228]** ⌈If the members/arguments itself are of type union, there shall be exactly one length field. The length field is added according to requirements [PRS_SOMEIP_00119], [PRS_SOMEIP_00121] with a size of 8,16 or 32 bit.⌋

**[PRS_SOMEIP_00229]** ⌈If the members/arguments itself are of type union, the length field shall cover the size of the type selector field, data and padding bytes.⌋

**Note:**
For the serialization without tags, the length field of unions does not cover the type selector field (see [PRS_SOMEIP_00126]). For the serialization with tags, it is required that the complete content of the serialized union is covered by the length field.

**[PRS_SOMEIP_00210]** ⌈A member of a non-extensible (standard) struct which is of type extensible struct, shall be serialized according to the requirements for extensible structs.⌋

**[PRS_SOMEIP_00211]** ⌈A member of an extensible struct which is of type non-extensible (standard) struct, shall be serialized according to the requirements for standard structs.⌋

**[PRS_SOMEIP_00222]** ⌈The alignment of variable length data according to [PRS_SOMEIP_00611] shall always be 8 bit.⌋

**Rationale:**
When alignment greater 8 bits is used, the serializer may add padding bytes after variable length data. The padding bytes are not covered by the length field. If the receiver does not know the Data ID of the member, it also does not know that it is variable length data and that there might be padding bytes.

**[PRS_SOMEIP_00241]** ⌈If TLV is used the size of the length field for arrays, structs, unions and strings shall be greater than 0.⌋

**Rationale:**
The TLV serialization requires the usage of length fields. When wire type 4 is used, the length field size must be statically configured. When wire types 5-7 (dynamic length

field size) are used, the static configuration of the length field size must also be present since not all length fields are preceded by a tag, e.g. structs contained in an array or the top-level struct contained in a SOME/IP event. Not using length fields here would result in ambiguities.

**[PRS_SOMEIP_00242]** ⌈The configured size of the length field for arrays, structs, unions and strings shall be identical.⌋

**Rationale:**
In case of an unknown member or argument, the deserializer cannot determine the actual datatype of the member/argument when wire type 4 is used.

**[PRS_SOMEIP_00243]** ⌈The size of the length field shall be configured for the top-level struct or method request/response. All arrays, unions, structs and strings used within a struct or all arguments within a method shall inherit the size of the length field from the top-level definition.⌋

**Rationale:**
In case of an unknown member or argument, the deserializer needs to know the size of the length field when wire type 4 is used. The easiest way is that the size of the length field is then only defined at the top-level element.

**[PRS_SOMEIP_00244]** ⌈Overriding the size of the length field at a subordinate array, union, struct or string or at an individual method argument shall not be allowed.⌋

**[PRS_SOMEIP_00216]** ⌈The serializer shall not include optional members/arguments in the serialized byte stream if they are marked as not available.⌋

**[PRS_SOMEIP_00220]** ⌈If the serialization with tags will be introduced for an existing service interface where tags have not been used, the major interface version shall be incremented and used to indicate this.⌋

**Note:**
The receiver only handles received messages that match all configured values of Message ID, Protocol Version, Interface Version and Message-Type (see [PRS_SOMEIP_00195]).

### Example for serializing structures with tags

```
struct myStruct
   unit8  a /* Data ID = 0 */
   uint8  b /* Data ID = 1 */
   struct c /* Data ID = 2 */
      uint8  c1
      uint8  c2
      struct c3
         uint8  c31
         struct c32
            uint8   c321
            uint8[] c322
            uint8   c323
         uint8  c33
      uint8[]c4
      uint8  c5
   uint8  d /* Data ID = 3 */
   struct e /* Data ID = 4 */
      uint8  e1
      uint8  e2
   uint8  f /* Data ID = 5 */
   uint8  g /* Data ID = 6 */
```

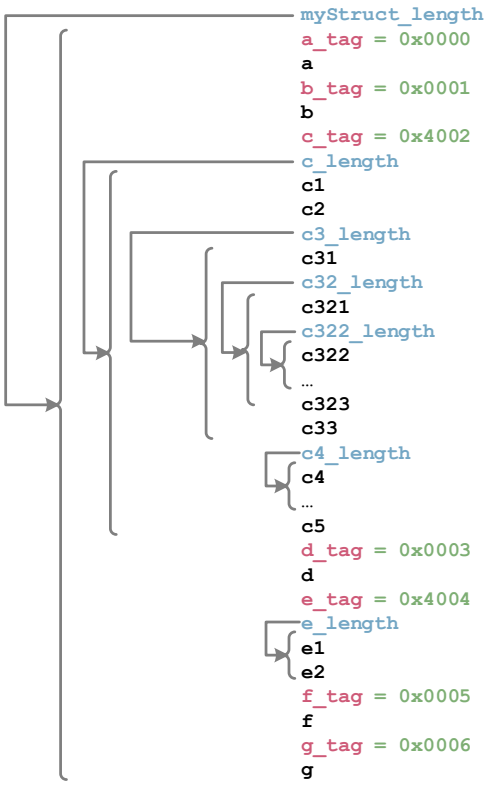


**Figure 5.3: Example 01 for serializing structures with tags**

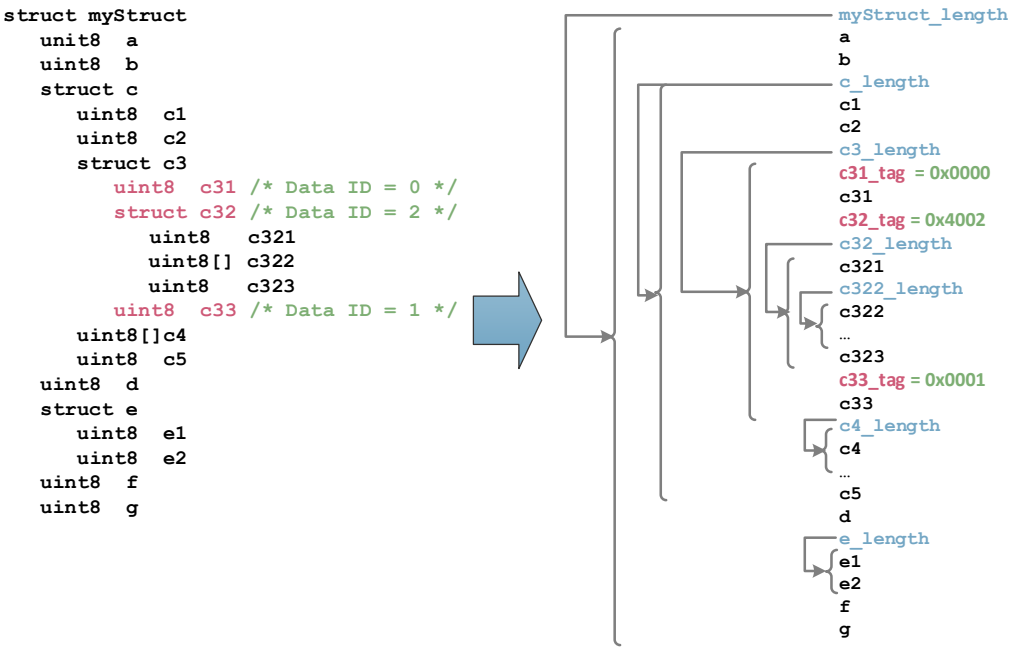


**Figure 5.4: Example 02 for serializing structures with tags**

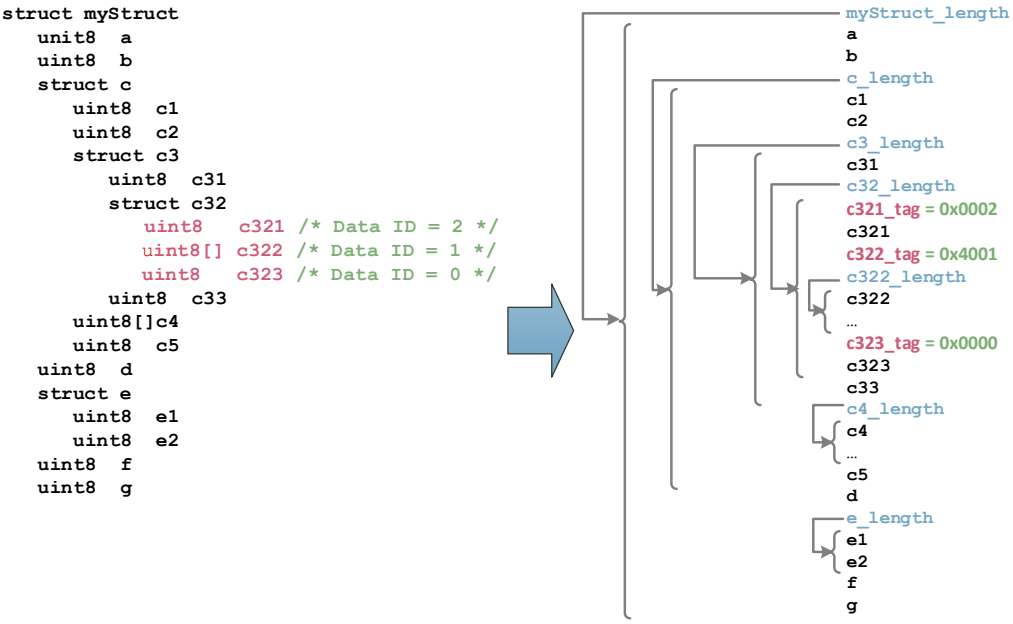


**Figure 5.5: Example 03 for serializing structures with tags**

**Example for serialization of arguments with tags**

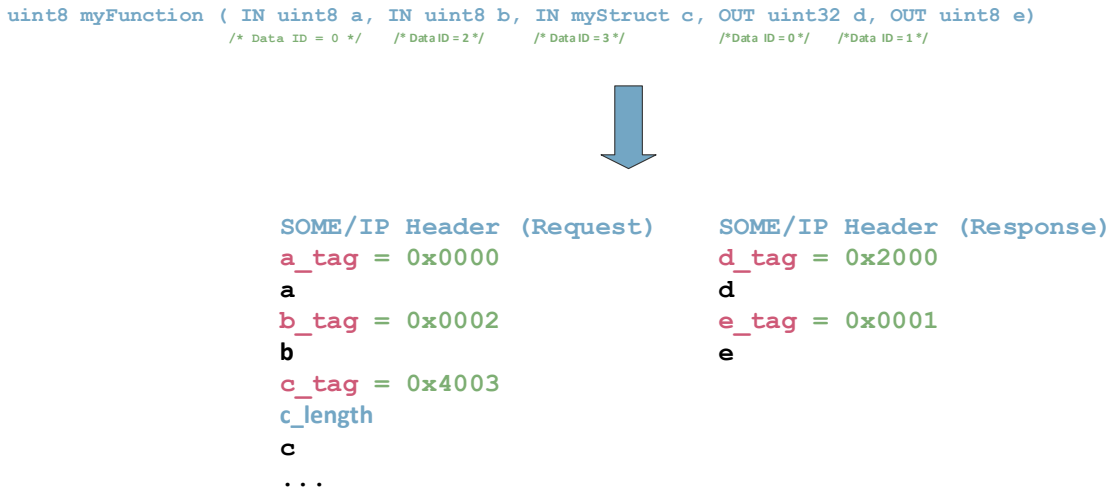


**Figure 5.6: Example for serialization of arguments with tags**

**Note:**
In the example Figure 5.6 there is no additional length field between the end of the SOME/IP header and the first tag. This would be redundant to the message length field in the SOME/IP header.

#### 5.1.3.4 Strings

Following requirements are common for both fixed length and dynamic length strings.

**[PRS_SOMEIP_00372]** ⌈Unicode encoding shall be determined by configuration parameter STRING_ENCODING.Different unicode encodings shall be supported according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004].If not configured explicitly via the configuration parameter STRING_ENCODING, the default value defined in [PRS_SOMEIP_00004] shall be used as the string encoding.⌋

**[PRS_SOMEIP_00948]** ⌈UTF-8 strings shall be zero terminated with a "\0" character. This means they shall end with a 0x00 Byte.⌋

**[PRS_SOMEIP_00084]** ⌈UTF-16LE and UTF-16BE strings shall be zero terminated with a "\0" character. This means they shall end with (at least) two 0x00 Bytes.⌋

**[PRS_SOMEIP_00085]** ⌈UTF-16LE and UTF-16BE strings shall have an even length.⌋

**[PRS_SOMEIP_00086]** ⌈UTF-16LE and UTF-16BE strings having an odd length the last byte shall be ignored.⌋

**[PRS_SOMEIP_00087]** ⌈All strings shall always start with a Byte Order Mark (BOM) in the first three (UTF-8) or two (UTF-16) bytes of the to be serialized array containing the string. The BOM shall be included in fixed-length-strings as well as dynamic-length strings. BOM allows the possibility to detect the used encoding.⌋

**[PRS_SOMEIP_00088] Allowed characters in string** ⌈SOME/IP shall support serialization of strings containing printable characters and the following control characters: BS (\b), HT (\t), LF (\n), VT (\v), FF (\f), CR (\r), ESC (\e).
⌋

**Note:**
Implementations are not required to explicitly check for the presence of these characters. The purpose of this requirement is to ensure consistent interoperability across platforms.

#### 5.1.3.4.1 Strings (fixed length)

**[PRS_SOMEIP_00760]** ⌈Fixed length strings may have a length field that is added in front of the string (for the sake of backwards compatible extension).The length of this length field shall be determined by the configuration parameter SIZE_OF_STRING_LENGTH_FIELD in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004].If not configured explicitly via the configuration parameter SIZE_OF_STRING_LENGTH_FIELD, the default value defined in [PRS_SOMEIP_00004] shall be used.⌋

**[PRS_SOMEIP_00373]** ⌈Strings shall be terminated with a "\0"-character despite having a fixed length.⌋

**[PRS_SOMEIP_00374]** ⌈The length of the string (this includes the "\0") in Bytes has to be specified in the data type definition.⌋

**[PRS_SOMEIP_00951] Padding requirement for strings** ⌈To transmit a fixed-length string whose actual content length is smaller than the fixed length defined in the interface definition, the sender shall pad the remaining bytes with the NUL character ('\0') until the configured fixed length is fully reached.⌋

#### 5.1.3.4.2 Strings (dynamic length)

**[PRS_SOMEIP_00089]** ⌈Strings with dynamic length shall start with a length field. The length is measured in Bytes.⌋

**[PRS_SOMEIP_00090]** ⌈The length field is placed before the BOM, and the BOM is included in the length.⌋

**[PRS_SOMEIP_00091]** ⌈String are terminated with a "\0".⌋

**Note:**
The maximum number of bytes of the string (including termination with "\0") shall also be derived from the data type definition.

**[PRS_SOMEIP_00092]** ⌈[PRS_SOMEIP_00084], [PRS_SOMEIP_00085] and [PRS_SOMEIP_00086] shall also be valid for strings with dynamic length.⌋

**[PRS_SOMEIP_00002] Length field for Dynamic length strings** ⌈Dynamic length strings shall have a length field that is added in front of the string. The length of this length field shall be determined by the configuration parameter SIZE_OF_STRING_LENGTH_FIELD in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter SIZE_OF_STRING_LENGTH_FIELD, the default value defined in [PRS_SOMEIP_00004] shall be used.⌋

**[PRS_SOMEIP_00095]** ⌈The length of the Strings length field is not considered in the value of the length field; i.e. the length field does not count itself.⌋

#### 5.1.3.5 Arrays

##### 5.1.3.5.1 Arrays (fixed length)

Fixed length arrays are easier for use in very small devices. Dynamic length arrays might need more resources on the ECU using them.

**[PRS_SOMEIP_00944]** ⌈Fixed length arrays may have a length field that is added in front of the array (for the sake of backwards compatible extension). The length of this length field shall be determined by the configuration parameter SIZE_OF_ARRAY_LENGTH_FIELD in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter SIZE_OF_ARRAY_LENGTH_FIELD, the default value defined in [PRS_SOMEIP_00004] shall be used.⌋

**Note:** Overruns of fixed-size arrays can only be detected with a length field.

**One-dimensional**

**[PRS_SOMEIP_00099]** ⌈The one-dimensional arrays with fixed length "n" shall carry exactly "n" elements of the same type. An optional length field may precede the first element (see [PRS_SOMEIP_00944].⌋

**Note:** If a length field is defined for a specific fixed-length array, then this array is represented on the bus as a composite of the length field and the collection of n elements of the same data type.

The layout of [PRS_SOMEIP_00099] is shown in Figure 5.7.

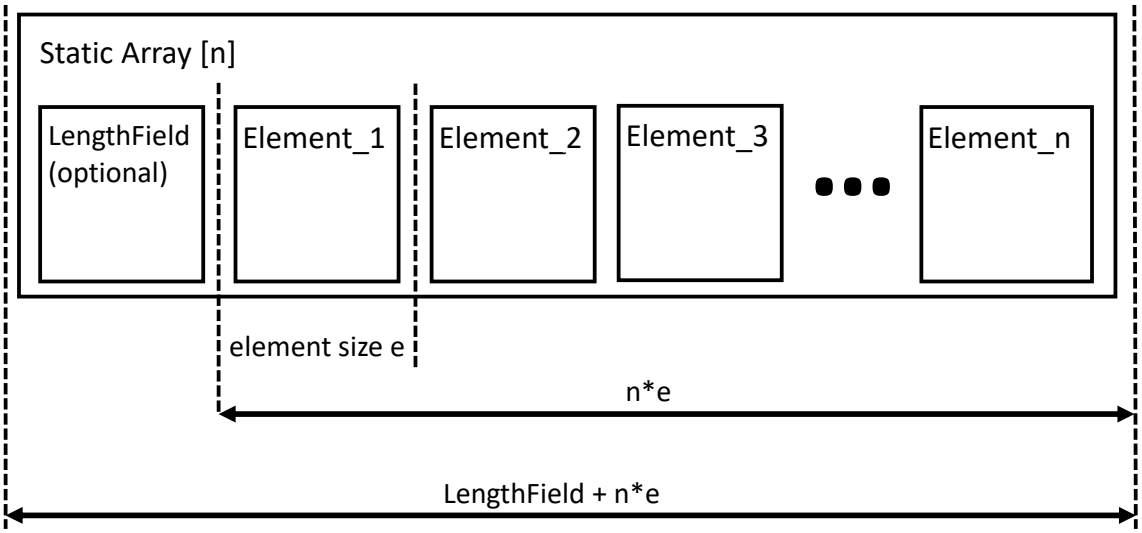


**Figure 5.7: One-dimensional array (fixed length)**

### Multidimensional

**[PRS_SOMEIP_00101]** ⌈The serialization of multidimensional arrays follows the in-memory layout of multidimensional arrays in the C/C++ programming language (row-major order).⌋

**Note:** If a length field is defined for a specific multidimensional fixed-length array, then this array is represented on the bus as a composite of a length field and n collections consisting each of a length field and m elements of the same data type.

The layout of [PRS_SOMEIP_00101] is shown in Figure 5.8.

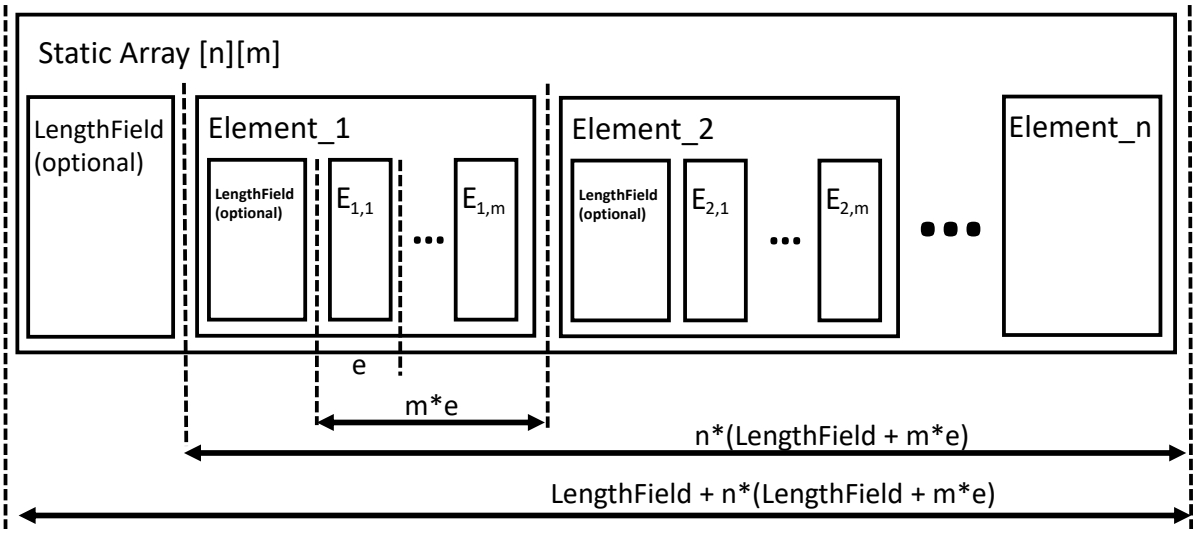


**Figure 5.8: Multidimensional array (fixed length)**

#### 5.1.3.5.2 Dynamic Length Arrays

**[PRS_SOMEIP_00375]** ⌈The layout of arrays with dynamic length shall be based on the layout of fixed length arrays.⌋

**[PRS_SOMEIP_00376]** ⌈A length field at the beginning of a dynamic length array shall be used to specify the length of the array in Bytes.⌋

**[PRS_SOMEIP_00377]** ⌈The length field of the dynamic length array shall describe the number of bytes this array occupies for `SOME/IP` transport including potential length fields and potential padding (according to [PRS_SOMEIP_00611]) of the individual array elements of the array but excluding the array's own length field.⌋

**Note:**
If the length of the length field is set to 0 Bits, the number of elements in the array has to be fixed; thus, being an array with fixed length.

The layout of dynamic arrays is shown in Figure 5.9 and Figure 5.10.

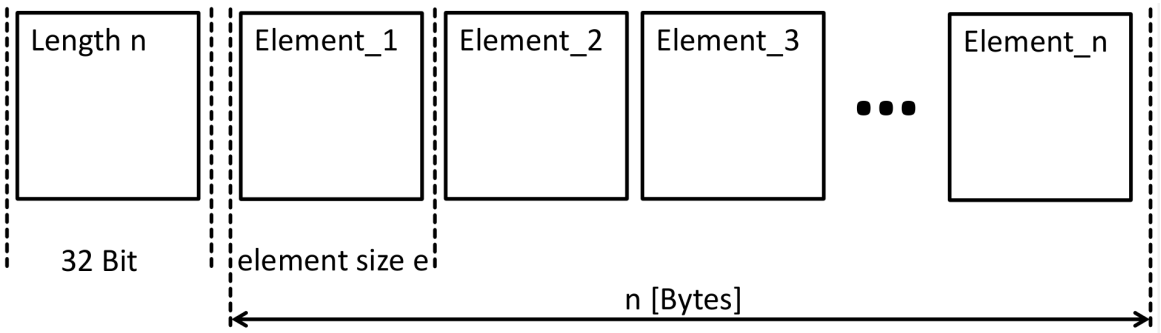


**Figure 5.9: One-dimensional array (dynamic length)**

In the one-dimensional array one length field is used, which carries the number of bytes used for the array.

The number of static length elements can be easily calculated by dividing by the size of an element.

In the case of dynamical length elements the number of elements cannot be calculated, but the elements must be parsed sequentially.

Figure 5.10 shows the structure of a Multidimensional Array of dynamic length.

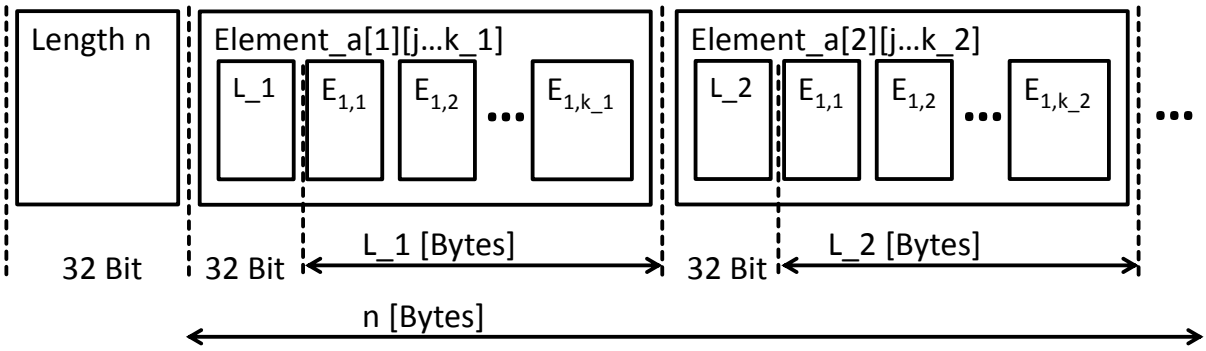


**Figure 5.10: Multidimensional array (dynamic length)**

**[PRS_SOMEIP_00114]** ⌈In multidimensional arrays every sub array of different dimensions shall have its own length field.⌋

If static buffer size allocation is required, the data type definition shall define the maximum length of each dimension.

Rationale: When measuring the length in Bytes, complex multi-dimensional arrays can be skipped over in deserialization.

SOME/IP also supports that different length for columns and different length for rows in the same dimension. See `k_1` and `k_2` in Figure 5.10. A length indicator needs to be present in front of every dynamic length array. This applies for both outer and all inner/nested arrays.

**[PRS_SOMEIP_00001] Length field for Dynamic length arrays** ⌈Dynamic length arrays shall have a length field that is added in front of the array. The length of this length field shall be determined by the configuration parameter SIZE_OF_ARRAY_LENGTH_FIELD in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter SIZE_OF_ARRAY_LENGTH_FIELD, the default value defined in [PRS_SOMEIP_00004] shall be used.⌋

#### 5.1.3.6 Enumeration

**[PRS_SOMEIP_00705]** ⌈Enumerations are not considered in SOME/IP. Enumerations shall be transmitted as unsigned integer datatypes.⌋

#### 5.1.3.7 Bitfield

**[PRS_SOMEIP_00300]** ⌈Bitfields shall be transported as unsigned datatypes uint8/uint16/uint32/uint64.⌋

The data type definition will be able to define the name and values of each bit.

#### 5.1.3.8 Union / Variant

There are use cases for defining data as unions on the network where the payload can be of different data types.

A union (also called variant) is such a parameter that can contain different types of data. For example, if one defines a union of type uint8 and type uint16, the union shall carry data which are a uint8 or a uint16.

Which data type will be transmitted in the payload can only be decided during execution. In this case, however, it is necessary to not only send the data itself but add an information about the applicable data type as a form of "meta-data" to the transmission.

By the means of the attached meta-data the sender can identify the applicable data type of the union and the receiver can accordingly access the data properly.

**[PRS_SOMEIP_00118]** ⌈A union shall be used to transport data with alternative data types over the network.⌋

**[PRS_SOMEIP_00119] Union (length field, type selector field and payload)** ⌈

| |
|---|
| Length field [32, 16, 8, 0 bit] |
| Type selector field [32, 16, 8 bit] |
| Payload including padding [length of padding = value of length field - actual payload length] |

⌋

**[PRS_SOMEIP_00126]** ⌈The length field of the union shall describe the number of bytes the payload of this union occupies for SOME/IP transport including potential length fields and potential padding (according to [PRS_SOMEIP_00611]) of the individual union elements but excluding the union's own length field and the type selector field.⌋

**Note:**
The padding can be used to align following data in the serialized data stream if configured accordingly.

**[PRS_SOMEIP_00121]** ⌈The length of the length field shall be defined by configuration parameter SIZE_OF_UNION_LENGTH_FIELD and shall be in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004].If not configured explicitly via the configuration parameter SIZE_OF_UNION_LENGTH_FIELD ,the default value defined in [PRS_SOMEIP_00004] shall be used as length of the length field that is added in front of the union.⌋

**[PRS_SOMEIP_00122]** ⌈A length of the length field of 0 Bit means that no length field will be written to the PDU.⌋

**[PRS_SOMEIP_00123]** ⌈If the length of the length field is 0 Bit, all types in the union shall be of the same length.⌋

**[PRS_SOMEIP_00129]** ⌈The type selector field shall specify the payload type of the payload.⌋

**[PRS_SOMEIP_00127]** ⌈The length of the type selector field shall be defined by configuration parameter SIZE_OF_UNION_TYPE_SELECTOR_FIELD and shall be in the allowed range according to the 'Allowed Range or Values' column in [PRS_SOMEIP_00004]. If not configured explicitly via the configuration parameter SIZE_OF_UNION_TYPE_SELECTOR_FIELD , the default value defined in [PRS_SOMEIP_00004] shall be used as length of the type selector field that is added in front of the union.⌋

**[PRS_SOMEIP_00906]** ⌈Possible values of the type selector field shall be defined by the configuration for each union separately.⌋

**[PRS_SOMEIP_00907]** ⌈The value 0 of the type selector field shall be reserved for the NULL type. In this case the length of the payload shall be 0.⌋

**Note:**
This denotes an empty union.

**[PRS_SOMEIP_00130]** ⌈The payload is serialized depending on the type in the type selector field.⌋

In the following example a length of the length field is specified as 32 Bits. The union shall support a uint8 and a uint16 as data. Both are padded to the 32 bit boundary (length=4 Bytes).

A uint8 will be serialized like shown in Table 5.6.

| Length = 4 Bytes | | | |
|---|---|---|---|
| Type = 1 | | | |
| uint8 | Padding 0x00 | Padding 0x00 | Padding 0x00 |

**Table 5.6: Example: uint8**

A uint16 will be serialized like shown in Table 5.7.

| Length = 4 Bytes | | |
|---|---|---|
| Type = 2 | | |
| uint16 | Padding 0x00 | Padding 0x00 |

**Table 5.7: Example: uint16**

### 5.1.4 De-serialization of Data Structures

The de-serialization process need to inspect the payload (serialized byte stream) of the received SOME/IP message. Thereby the de-serialization process need to identify the elements within the received byte stream and compare the identified elements with the configured data type(s) of the corresponding service interface (please note, the data type is derived from the interface specification, which defines the exact position of all data structures in a SOME/IP message). The possibility to identify elements in a dedicated SOME/IP serialized byte stream depend on the interface specification and the serialization properties. The serialization properties define among others:

- if structured data types are serialized with a length field in front
- if tag-length-value are used for encoding, which include data ids and the possibility specify optional data members

The de-serialization process of a SOME/IP messages need to consider the received message length and deal with a message length which may be larger than expected according the interface specification. This is needed to support backward compatible communication, where ECUs of a heterogeneous in-vehicle network (re-used ECUs and new developed ECUs) communicate via SOME/IP serialized byte streams. Note that the feature of "complementary default value during reception of less data than expected" is no longer supported by AUTOSAR. The subsequential chapters describe the expected behavior of the de-serialization process.

#### 5.1.4.1 Structured DataTypes (structs)

If more data than expected is received, the de-serialization process accepts all received elements of the SOME/IP message payload that correspond to the configured service interface data type and skips unknown elements of the de-serialized SOME/IP message payload. If less data than expected is received, the de-serialization process is aborted and the message is treated as malformed.

**[PRS_SOMEIP_00371]** ⌈If the length is greater than the length of the struct as specified in the data type definition only the bytes specified in the data type shall be interpreted and the other bytes shall be skipped based on the length field.⌋

**[PRS_SOMEIP_00900]** ⌈If the length is less than the sum of the lengths of all struct members, then the deserialization shall be aborted and the message shall be treated as malformed.⌋

#### 5.1.4.2 Structured Datatypes and Arguments with Identifier and optional members ('TLV')

If the de-serialization process detect an unknown optional member according the configured service interface data type, then the de-serialization process should ignore this member. If the de-serialization process detect a missing member/argument, which is required according the configured service interface data type, then the SOME/IP message is treated as malformed.

**[PRS_SOMEIP_00223]** ⌈The deserializer shall ignore optional members/arguments which are not available in the serialized byte stream.⌋

**[PRS_SOMEIP_00217]** ⌈If the deserializer reads an unknown Data ID (i.e. not contained in its data definition), it shall skip the unknown member/argument by using the information of the wire type and length field.⌋

**[PRS_SOMEIP_00209] Behaviour if a required member/argument is missing in the received SOME/IP message** ⌈If the deserializer cannot find a required (i.e. non-optional) member/argument defined in its data definition in the serialized byte stream, the deserialization shall be aborted and the message shall be treated as malformed.⌋

**[PRS_SOMEIP_00384] Behaviour if a required member/argument is found multiple times in the received SOME/IP message** ⌈If the deserializer finds a member/argument defined in its data definition in the serialized byte stream multiple times, then the deserialization shall be aborted and the message shall be treated as malformed.⌋

**[PRS_SOMEIP_00383] Behaviour if a required member/argument is found with invalid wire type in the received SOME/IP message** ⌈If the deserializer finds a required (i.e. non-optional) member/argument defined in its data definition in the serial-

ized byte stream with invalid wire type, then the deserialization shall be aborted and the message shall be treated as malformed.⌋

#### 5.1.4.3 Strings

Strings could be configured with a fixed length or dynamic length. Independent a string has fixed length or dynamic length configured, a serialized string, which is received with a larger length than expected according the configured service interface data type, should be treated as malformed. For strings with fixed length and received with a length which is less than expected and this string is correctly terminated, the string should be processed. Otherwise the message shall be treated as malformed.

##### 5.1.4.3.1 Strings (fixed length)

**[PRS_SOMEIP_00911] Handling of Fixed-Length Strings exceeding defined size** ⌈If a length field is present for a received string, and it indicates a size greater than the fixed-length defined in the data type, the deserialization shall be aborted and the message shall be treated as malformed.
The receiver shall check whether the `NUL` character ('\0') appears within the range defined by the length field (if present) or the fixed-length defined in the data type. If the `NUL` character does not appear within this range, the deserialization shall be aborted and the message shall be treated as malformed.⌋

**[PRS_SOMEIP_00912] Handling of Fixed-Length Strings with reduced size** ⌈If the length of a received string with fixed length(defined by the position of the first `NUL` character) is less than expected (expectation shall be based on the data type definition) and it is correctly terminated using '\0', it shall be accepted.The next valid data item following the fixed-length string shall always be interpreted as starting at the position given by:

- the length field (if present), or
- the fixed-length defined in the data type definition (if no length field exists).

⌋

**[PRS_SOMEIP_00913]** ⌈If the length of a string with fixed length is less than expected (expectation shall be based on the data type definition) and it is not correctly terminated using "\0", the deserialization shall be aborted and the message shall be treated as malformed.⌋

Instead of transferring application strings as SOME/IP strings with BOM and "\0" termination, strings can also be transported as plain dynamic length arrays without BOM and "\0" termination (see chapter 5.1.3.5.2). Please note that this requires the full string handling (e.g. endianness conversion) to be done in the applications.

#### 5.1.4.3.2 Strings (dynamic length)

**[PRS_SOMEIP_00914]** ⌈If the length of a string with variable length is greater than expected (expectation shall be based on the data type definition), the deserialization shall be aborted and the message shall be treated as malformed.⌋

### 5.1.4.4 Arrays

Arrays could be configured with a fixed length or dynamic length. Independent if an array has fixed length or dynamic length configured, if a serialized array was received with a larger length than expected according the configured service interface data type, then all known elements according the configured service interface data type should be considered and the remaining elements should be skipped by the de-serialization process.

For arrays with fixed length and the de-serialization process detect a missing element, which is required according the configured service interface data type, then the SOME/IP message is treated as malformed.

#### 5.1.4.4.1 Arrays (fixed length)

**[PRS_SOMEIP_00917]** ⌈If the length of a fixed length array is greater than expected (expectation shall be based on the data type definition) only the elements specified in the data type shall be interpreted and the other bytes shall be skipped based on the length field.⌋

**[PRS_SOMEIP_00207] Behaviour if a required element is missing in the received SOME/IP message** ⌈If the length of a fixed length array is less than expected (expectation shall be based on the data type definition) and no substitution for the missing data can be provided locally by the receiver, the deserialization shall be aborted and the message shall be treated as malformed.⌋

**Note:** Overruns of fixed-size arrays can only be detected with a length field.

#### 5.1.4.4.2 Arrays (dynamic length)

**[PRS_SOMEIP_00919]** ⌈If the length of a variable length array is greater than expected (expectation shall be based on the data type definition) only the elements specified in the data type shall be interpreted and the other bytes shall be skipped based on the length field.⌋

#### 5.1.4.5 Enumeration

No further requirements considered for the deserialization.

#### 5.1.4.6 Bitfield

No further requirements considered for the deserialization.

#### 5.1.4.7 Union / Variant

**[PRS_SOMEIP_00915]** ⌈If the length of a union is greater than expected (expectation shall be based on the data type definition) only the bytes specified in the data type shall be interpreted and the other bytes shall be skipped based on the length field.⌋

**[PRS_SOMEIP_00916]** ⌈If the length of a union is less than expected (expectation shall be based on the data type definition) it shall depend on the inner data type whether valid data can be deserialized or the deserialization shall be aborted and the message shall be treated as malformed.⌋

## 5.2 Specification of SOME/IP Protocol

This chapter describes the Remote Procedure Call(RPC), Event Notifications and Error Handling of SOME/IP.

### 5.2.1 Transport Protocol Bindings

In order to transport SOME/IP messages different transport protocols may be used. SOME/IP currently supports UDP and TCP. Their bindings are explained in the following sections, while Chapter 7 discusses which transport protocol to choose.

**[PRS_SOMEIP_00138]** ⌈If a server runs different instances of the same service, messages belonging to different service instances shall be mapped to the service instance by the transport protocol port on the server side.⌋

For details of see Chapter 5.2.1.3

**[PRS_SOMEIP_00535]** ⌈All Transport Protocol Bindings shall support transporting more than one SOME/IP message in a Transport Layer PDU (i.e. UDP packet or TCP segment).⌋

**[PRS_SOMEIP_00142]** ⌈The receiving SOME/IP implementation shall be capable of receiving unaligned SOME/IP messages transported by UDP or TCP.⌋

Rationale:
When transporting multiple SOME/IP payloads in UDP or TCP the alignment of the payloads can be only guaranteed, if the length of every payloads is a multiple of the alignment size (e.g. 32 bits).

**[PRS_SOMEIP_00140]** ⌈The header format allows transporting more than one SOME/IP message in a single packet. The SOME/IP implementation shall identify the end of a SOME/IP message by means of the SOME/IP length field. Based on the packet length field, SOME/IP shall determine if there are additional SOME/IP messages in the packet. This shall apply for UDP and TCP transport.⌋

**[PRS_SOMEIP_00141]** ⌈Each SOME/IP payload shall have its own SOME/IP header.⌋

**[PRS_SOMEIP_00940]** ⌈One Service-Instance can use the following setup for its communication of all the methods, events, and notifications:

- up to one TCP connection
- up to one UDP unicast connection
- up to one UDP multicast connection

⌋

#### 5.2.1.1 UDP Binding

**[PRS_SOMEIP_00139]** ⌈The UDP binding of SOME/IP shall be achieved by transporting SOME/IP messages in UDP packets.⌋

**[PRS_SOMEIP_00137]** ⌈SOME/IP protocol shall not restrict the usage of UDP fragmentation.⌋

**[PRS_SOMEIP_00943]** ⌈The client and server shall use a single UDP unicast connection for all methods, events, and notifications of a Service-Instance which are configured to be communicated using UDP unicast.⌋

**[PRS_SOMEIP_00942]** ⌈The client and server shall use a single UDP multicast address combination ("connection") per eventgroup, which is configured to be communicated using UDP multicast.

If the same multicast address is shared between different service instances of the same service, then the port number of the UDP multicast address combination used for each of these service instances shall be different, at least on server side.⌋

Note: Please refer to [PRS_SOMEIP_00163] and the rational below [PRS_SOMEIP_00163] for detailed information.

#### 5.2.1.2 TCP Binding

The TCP binding of SOME/IP is heavily based on the UDP binding. In contrast to the UDP binding, the TCP binding allows much bigger SOME/IP messages and uses the robustness features of TCP (coping with loss, reorder, duplication, etc.).

In order to lower latency and reaction time, Nagle's algorithm should be turned off (TCP_NODELAY).

**[PRS_SOMEIP_00706]** ⌈When the TCP connection is lost, pending requests shall be handled if a timeout occurred.⌋

Since TCP handles reliability, additional means of reliability are not needed.

**[PRS_SOMEIP_00707]** ⌈The client and server shall use a single TCP connection for all methods, events, and notifications of a Service-Instance which are configured to be communicated using TCP.⌋

**[PRS_SOMEIP_00708]** ⌈The TCP connection shall be opened by the client, when the first method call shall be transported or the client tries to receive the first notifications.⌋

The client is responsible for re-establishing the TCP connection whenever it fails.

**[PRS_SOMEIP_00709]** ⌈The TCP connection shall be closed by the client, when the TCP connection is not required anymore.⌋

**[PRS_SOMEIP_00710]** ⌈The TCP connection shall be closed by the client, when all Services using the TCP connections are not available anymore (stopped or timed out).⌋

**[PRS_SOMEIP_00711]** ⌈The server shall not stop the TCP connection when stopping all services. Give the client enough time to process the control data to shutdown the TCP connection itself.⌋

Rational:
When the server closes the TCP connection before the client recognized that the TCP is not needed anymore, the client will try to reestablish the TCP connection.

**Allowing resync to TCP stream using Magic Cookies**

**[PRS_SOMEIP_00154]** ⌈In order to allow testing tools to identify the boundaries of SOME/IP Message transported via TCP, the SOME/IP Magic Cookie Message may be inserted into the SOME/IP messages over TCP message stream at regular distances.⌋

**[PRS_SOMEIP_00160]** ⌈The layout of the Magic Cookie Messages shall consist of the following fields:

- for communication from Client to Server:

  - Message ID (Service ID/Method ID): `0xFFFF 0000`
  - Length: `0x0000 0008`
  - Request ID (Client ID/Session ID): `0xDEAD BEEF`
  - Protocol Version: `0x01`
  - Interface Version: `0x01`
  - Message Type: `0x01`
  - Return Code: `0x00`
- for communication from Server to Client:
  - Message ID (Service ID/Method ID): `0xFFFF 8000`
  - Length: `0x0000 0008`
  - Request ID (Client ID/Session ID): `0xDEAD BEEF`
  - Protocol Version: `0x01`
  - Interface Version: `0x01`
  - Message Type: `0x02`
  - Return Code: `0x00`

⌋

The layout of the Magic Cookie Messages is shown in Figure 5.11.

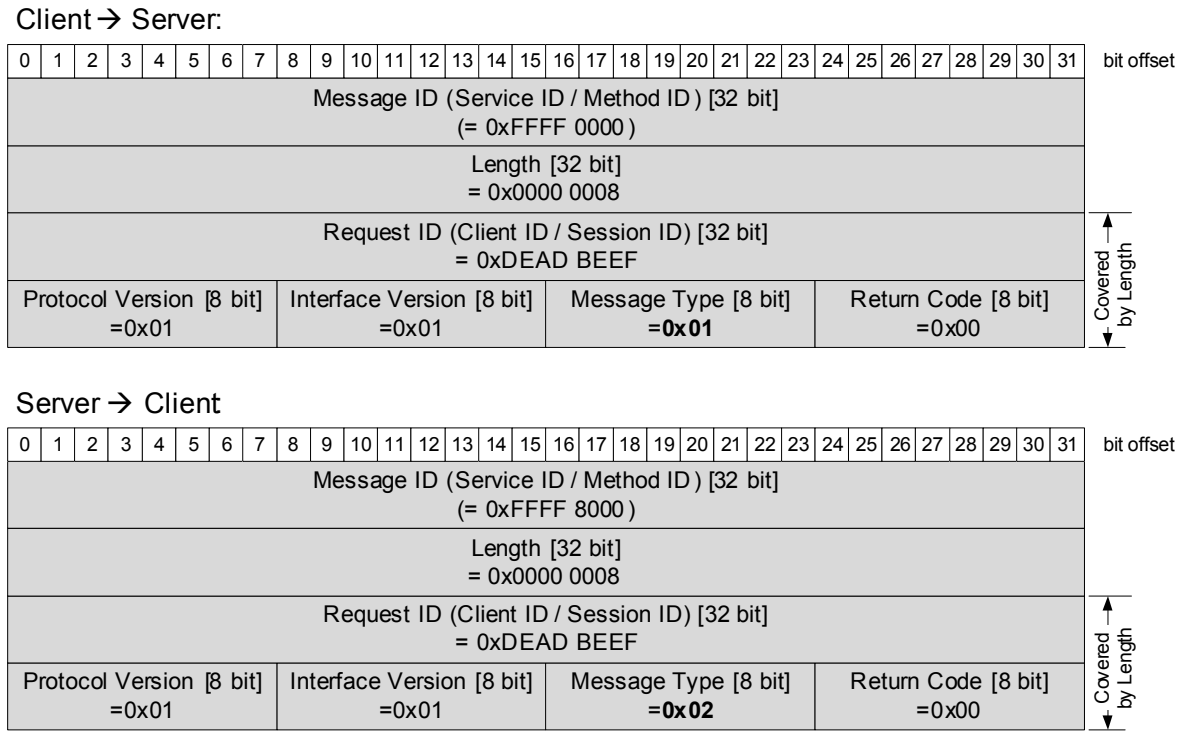


**Figure 5.11: SOME/IP Magic Cookie Message for SOME/IP**

#### 5.2.1.3 Multiple Service-Instances

**[PRS_SOMEIP_00162]** ⌈Service-Instances of the same Service are identified through different Instance IDs. It shall be supported that multiple Service-Instances reside on different ECUs as well as multiple Service-Instances of one or more Services reside on one single ECU.⌋

**[PRS_SOMEIP_00163]** ⌈While several Service-Instances of different Services shall be able to share the same port number of the transport layer protocol used on both the provided/server and the consumed/client side, multiple Service-Instances of the same Service on the provided/server side on one single ECU shall use different port numbers per Service-Instance. Multiple Service-Instances of the same Service on the required/client side on one single ECU may use the same port number per Service-Instance.⌋

Rationale: Normal SOME/IP (not SOME/IP-SD) messages do not carry the Service-Instance ID as a dedicated field in the SOME/IP header. - Thus port numbers (and perhaps the transport protocol) need to be used to distinguish **different Service-Instance** of the **same Service** of a single ECU. This way a Service-Instance can be identified through the combination of the Service ID combined with the endpoint information (i.e., IP-address, transport protocol (UDP/TCP), and port number). It is sufficient to use different port numbers for the different Service-Instances of the same Service on **either**

**the server or the client side**, since only a single difference in the 4-tuple <src IP, src port, dst IP, dst port > is sufficient as a distinguishing criterion. As the server is the one actually **providing** the different Service-Instances, the server is also the natural place to handle the distinction. = > **The server shall use different port numbers for providing different Service-Instances of the same Service.**

Recommendation: It is recommended that instances use the same port number for UDP and TCP. If a Service-Instance uses UDP port x, only this Service-Instance of the Service and not another Service-Instance of the same Service should use exactly TCP port x for its Service provision.

#### 5.2.1.4 Transporting large SOME/IP messages of UDP (SOME/IP-TP)

The UDP binding of SOME/IP can only transport SOME/IP messages that fit directly into an IP packet. If larger SOME/IP messages need to be transported over UDP (e.g. of 32 KB) the SOME/IP Transport Protocol (SOME/IP-TP) shall be used. The SOME/IP message too big to be transported directly with the UDP binding shall be called "original" SOME/IP message. The "pieces" of the original SOME/IP message payload transported in SOME/IP-TP messages shall be called "segments".

Use TCP only if very large chunks of data need to be transported (> 1400 Bytes) and no hard latency requirements in the case of errors exists

**[PRS_SOMEIP_00720]** ⌈SOME/IP messages using SOME/IP-TP shall activate Session Handling (Session ID must be unique for the original message).⌋

**[PRS_SOMEIP_00721]** ⌈All SOME/IP-TP segments shall carry the Session ID of the original message; thus, they have all the same Session-ID.⌋

**[PRS_SOMEIP_00722]** ⌈SOME/IP-TP segments shall have the TP-Flag of the Message Type set to 1.⌋

**[PRS_SOMEIP_00723]** ⌈SOME/IP-TP segments shall have a TP header right after the SOME/IP header (i.e. before the SOME/IP payload) with the following structure (bits from highest to lowest):

- Offset [28 bits]
- Reserved Flag [1 bit]
- Reserved Flag [1 bit]
- Reserved Flag [1 bit]
- More Segments Flag [1 bit]

⌋

SOME-IP-TP-Header is as shown in Table 5.8.

| 0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 |
|---|---|---|---|
| Message ID (Service ID / Method ID) [32 Bit] | | | |
| Length [32 Bit] | | | |
| Request ID (Client ID / Session ID) [32 Bit] | | | |
| Protocol Version [8 Bit] | Interface version [8 Bit] | **Message Type [8 Bit]** | Return Code [8 Bit] |
| Offset [28 bit] | | | RES 0 0 0 / M |
| Payload [variable size] | | | |

**Table 5.8: SOME/IP TP header**

**[PRS_SOMEIP_00931]** ⌈SOME/IP-TP Header shall be encoded in network byte order (big endian).⌋

**[PRS_SOMEIP_00724]** ⌈The Offset field shall transport the upper 28 bits of a uint32. The lower 4 bits shall be always interpreted as 0.⌋

**Note:**
This means that the offset field can only transport offset values that are multiples of 16 bytes.

**[PRS_SOMEIP_00725]** ⌈The Offset field of the TP header shall be set to the offset in bytes of the transported segment in the original message.⌋

**[PRS_SOMEIP_00726]** ⌈The Reserved Flags shall be set to 0 by the sender and shall be ignored (and not checked) by the receiver.⌋

**[PRS_SOMEIP_00727]** ⌈The More Segments Flag shall be set to 1 for all segments but the last segment. For the last segment it shall be set to 0.⌋

**[PRS_SOMEIP_00728]** ⌈The SOME/IP length field shall be used as specified before. This means it covers the first 8 bytes of the SOME/IP header and all bytes after that.⌋

**Note:**
This means that for a SOME/IP-TP message transporting a segment, the SOME/IP length covers 8 bytes of the SOME/IP header, the 4 bytes of the TP header, and the segment itself.

**[PRS_SOMEIP_00729]** ⌈The length of a segment must reflect the alignment of the next segment based on the offset field. Therefore, all but the last segment shall have a length that is a multiple of 16 bytes.⌋

**[PRS_SOMEIP_00730] Maximum segement length** ⌈In accordance with the recommended payload size (see [PRS_SOMEIP_00382]), the maximum length of a segment that is correctly aligned should be 1392 bytes.⌋

**[PRS_SOMEIP_00731]** ⌈SOME/IP-TP messages shall use the same Message ID (i.e. Service ID and Method ID), Request ID (i.e. Client ID and Session ID), Protocol Version, Interface Version, and Return Code as the original message.⌋

**Note:**
As described above the Length, Message Type, and Payload are adapted by SOME/IP-TP.

### Example

This example describes how an original SOME/IP message of 5880 bytes payload has to be transmitted. The Length field of this original SOME/IP message is set to 8 + 5880 bytes.

| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | | | |
|---|---|---|---|
| Message ID (Service ID / Method ID) [32 Bit]<br>0x0101 0009 | | | |
| Length [32 Bit]<br>= 8 + 5880 | | | |
| Request ID (Client ID / Session ID) [32 Bit]<br>0x0001 0005 | | | |
| Protocol Version [8 Bit]<br>0x01 | Interface version [8 Bit]<br>0x01 | Message Type [8 Bit]<br>b00000000 [0x00] | Return Code [8 Bit]<br>0x00 |
| Payload [5880 Bytes]<br>0x00 0x00 0x30 0x00<br>0x00 0x01 0x02 0x03<br>... | | | |

**Table 5.9: Example: Header of Original SOME/IP message**

This original SOME/IP message will now be segmented into 5 consecutive SOME/IP segments. Every payload of these segments carries at most 1392 bytes in this example.

For these segments, the SOME/IP TP module adds additional TP fields (marked red). The Length field of the SOME/IP carries the overall length of the SOME/IP segment including 8 bytes for the Request ID, Protocol Version, Interface Version, Message Type and Return Code. Because of the added TP fields (4 bytes), this Length information is extended by 4 additional SOME/IP TP bytes.

The following figure provides an overview of the relevant SOME/IP header settings for every SOME/IP segment:

| | Length (Bytes) | Message Type [TP-Flag] | Offset Value | More Segment Flag |
|---|---|---|---|---|
| 1st segment | 8 + 4 + 1392 = 1404 | TP-Flag = '1' | 0 | 1 |
| 2nd segment | 8 + 4 + 1392 = 1404 | TP-Flag = '1' | 87 | 1 |
| 3rd segment | 8 + 4 + 1392 = 1404 | TP-Flag = '1' | 174 | 1 |
| 4th segment | 8 + 4 + 1392 = 1404 | TP-Flag = '1' | 261 | 1 |
| 5th segment | 8 + 4 + 312 = 324 | TP-Flag = '1' | 348 | 0 |

**Table 5.10: Example: Overview of relevant SOME/IP TP headers**

**Note:**
Please be aware that the value provided within the Offset Field is given in units of 16 bytes, i.e.: The Offset Value of 87 correspond to 1392 bytes Payload.

The complete SOME/IP headers of the SOME/IP segments message will look like this in detail:

- The first 4 segments contain 1392 Payload bytes each with "More Segments Flag" set to '1':

<table>
<tr><td colspan="32">0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td></tr>
<tr><td colspan="32">Message ID (Service ID / Method ID) [32 Bit]<br>0x0101 0009</td></tr>
<tr><td colspan="32">Length [32 Bit]<br>8+4+1392 (1404)</td></tr>
<tr><td colspan="32">Request ID (Client ID / Session ID) [32 Bit]<br>0x0001 0005</td></tr>
<tr><td colspan="8">Protocol Version [8 Bit]<br>0x01</td><td colspan="8">Interface version [8 Bit]<br>0x01</td><td colspan="8">Message Type [8 Bit]<br>b00100000 [0x20]</td><td colspan="8">Return Code [8 Bit]<br>0x00</td></tr>
<tr><td colspan="28">Offset [28 bit]</td><td colspan="3">RES<br>0 0 0</td><td>M<br>1</td></tr>
<tr><td colspan="32">Payload [1392 Bytes]<br>0x00 0x17 0x28 0x33<br>0x28 0x03 0x18 0x32<br>...</td></tr>
</table>

**Table 5.11: Example: Header of the SOME/IP segments**

- The last segment (i.e. #5) contains the remaining 312 Payload bytes of the original 5880 bytes payload. This last segment is marked with "More Segments flag" set to '0'.

| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | | | |
|---|---|---|---|
| Message ID (Service ID / Method ID) [32 Bit]<br>0x0101 0009 | | | |
| Length [32 Bit]<br>8+4+312 (324) | | | |
| Request ID (Client ID / Session ID) [32 Bit]<br>0x0001 0005 | | | |
| Protocol Version [8 Bit]<br>0x01 | Interface version [8 Bit]<br>0x01 | **Message Type [8 Bit]**<br>b00100000 [0x20] | Return Code [8 Bit]<br>0x00 |
| Offset [28 bit] | | RES<br>0 0 0 | M<br>0 |
| Payload [312 Bytes]<br>...<br>0x4d 0x65 0x67 0x61<br>0x6e 0x46 0x6f 0x78 | | | |

**Table 5.12: Example: Header of the last SOME/IP segments**

### Sender specific behavior

**[PRS_SOMEIP_00732]** ⌈The sender shall segment only messages that were configured to be segmented.⌋

**[PRS_SOMEIP_00733]** ⌈The sender shall send segments in ascending order.⌋

**[PRS_SOMEIP_00734]** ⌈The sender shall segment in a way that all segments with the More Segment Flag set to 1 are of the same size.⌋

**[PRS_SOMEIP_00735]** ⌈The sender shall try to maximize the size of segments within limitations imposed by this specification.⌋

**[PRS_SOMEIP_00736]** ⌈The sender shall not send overlapping or duplicated segments.⌋

### Receiver specific behavior

**[PRS_SOMEIP_00738]** ⌈The receiver shall match segments for reassembly based on the configured values of Message-ID, Protocol-Version, Interface-Version and Message-Type (w/o TP Flag).⌋

**[PRS_SOMEIP_00740]** ⌈It shall be supported to reassemble multiple messages with the same Message ID but sent from different clients (difference in Sender IP, Sender Port, or Client ID) in parallel. This should be controlled by configuration and determines the amount of "reassembly buffers".⌋

**[PRS_SOMEIP_00741]** ⌈The Session ID shall be used to detect the next original message to be reassembled.⌋

**[PRS_SOMEIP_00742]** ⌈The receiver shall start a new reassembly (and may throw away old segments that were not successfully reassembled), if a new segment with a different Session-ID is received.⌋

**[PRS_SOMEIP_00743]** ⌈The receiver should only reassemble up to its configured buffer size and skip the rest of the message.⌋

**[PRS_SOMEIP_00744]** ⌈Only correctly reassembled message of up to the configured size shall be passed to an application.⌋

**Note:**
This means that the implementation must make sure that all bytes of the message must be bytes that were received and reassembled correctly. Counting non-overlapping, non-duplicated bytes and comparing this to the length could be a valid check.

**[PRS_SOMEIP_00379]** . ⌈The receiver shall cancel the current assembly process, when no segment has been received during a configured `SOMEIPTP_REASSEMBLY_TIMEOUT` (see [PRS_SOMEIP_00378]) period.⌋

**[PRS_SOMEIP_00745]** ⌈The Return Code of the last segment used for reassembly shall be used for the reassembled message.⌋

**[PRS_SOMEIP_00746]** ⌈During reassembling the SOME/IP TP segments into a large unsegmented message, the Message Type shall be adapted, the TP Flag shall be reset to 0.⌋

**[PRS_SOMEIP_00747]** ⌈The receiver shall support reassembly of segments that are received in ascending and descending order.⌋

**[PRS_SOMEIP_00749]** ⌈When a missing segment is detected during assembly of a SOME/IP message, the current assembly process shall be canceled.⌋

**Note:**
This means that reordering is not supported.

**[PRS_SOMEIP_00750]** ⌈Interleaving of different segmented messages using the same buffer (e.g. only the Session-ID and payload are different) is not supported.⌋

**Note:**
This prohibits that equal events (same Message-ID, IP-Addresses, ports numbers, and transport protocol) arrive in the wrong order, when some of their segments get reordered.

**[PRS_SOMEIP_00751]** ⌈Reordering of segments of completely different original messages (e.g. Message ID is different) is not of concern since those segments go to different buffers.⌋

**[PRS_SOMEIP_00752]** ⌈The receiver shall correctly reassemble overlapping and duplicated segments by overwriting based on the last received segment.⌋

**[PRS_SOMEIP_00753]** ⌈The receiver may cancel reassembly, if overlapping or duplicated segments change already written bytes in the buffer, if this feature can be turned off by configuration.⌋

**[PRS_SOMEIP_00754]** ⌈The receiver shall be able to detect and handle obvious errors gracefully. E.g. cancel reassembly if segment length of a segment with MS=1 is not a multiple of 16.⌋

**Note:**
This means that buffer overflows or other malfunction shall be prevented by the receiving code.

### 5.2.2 Request/Response Communication

One of the most common communication patterns is the request/response pattern. One communication partner (Client) sends a request message, which is answered by another communication partner (Server).

**[PRS_SOMEIP_00920]** ⌈For the SOME/IP request message the client has to do the following for payload and header:

- Construct the payload
- Set the Message ID based on the method the client wants to call
- Set the Length field to 8 bytes (for the part of the SOME/IP header after the length field) + length of the serialized payload
- Optionally set the Request ID to a unique number (shall be unique for client only)
- Set the Protocol Version according [PRS_SOMEIP_00052]
- Set the Interface Version according to the interface definition
- Set the Message Type to `REQUEST` (i.e. `0x00`)
- Set the Return Code to `0x00`

⌋

**[PRS_SOMEIP_00921]** ⌈To construct the payload of a request message, all input or inout arguments of the method shall be serialized according to the order of the arguments within the signature of the method.⌋

**[PRS_SOMEIP_00922]** ⌈The server builds the header of the response based on the header of the client's request and does in addition:

- Construct the payload
- take over the Message ID from the corresponding request
- Set the length to the 8 Bytes + new payload size
- take over the Request ID from the corresponding request
- Set the Message Type to `RESPONSE` (i.e. `0x80`) or `ERROR` (i.e. `0x81`)
- set Return Code to a return code according to [PRS_SOMEIP_00191].

⌋

**[PRS_SOMEIP_00923]** ⌈To construct the payload of a response message, all output or inout arguments of the method shall be serialized according to the order of the arguments within the signature of the method.⌋

**[PRS_SOMEIP_00927]** ⌈A server shall not sent a response message for a request with a specific Request ID until the corresponding request message has been received.⌋

**[PRS_SOMEIP_00928]** ⌈A client shall ignore the reception of a response message with a specific Request ID, when the corresponding request message has not yet been sent completely.⌋

### 5.2.3 Fire&Forget Communication

Requests without response message are called fire&forget.

**[PRS_SOMEIP_00924]** ⌈For the SOME/IP request-no-return message the client has to do the following for payload and header:

- Construct the payload
- Set the Message ID based on the method the client wants to call
- Set the Length field to 8 bytes (for the part of the SOME/IP header after the length field) + length of the serialized payload
- Optionally set the Request ID to a unique number (shall be unique for client only)
- Set the Protocol Version according [PRS_SOMEIP_00052]
- Set the Interface Version according to the interface definition
- Set the Message Type to `REQUEST_NO_RETURN` (i.e. `0x01`)
- Set the Return Code to `0x00`

⌋

**[PRS_SOMEIP_00171]** ⌈Fire & Forget messages shall not return an error. Error handling and return codes shall be implemented by the application when needed.⌋

### 5.2.4 Notification Events

Notifications describe a general Publish/Subscribe-Concept. Usually the server publishes a service to which a client subscribes. On certain cases the server will send the client an event, which could be for example an updated value or an event that occurred.

SOME/IP is used only for transporting the updated value and not for the publishing and subscription mechanisms. These mechanisms are implemented by SOME/IP-SD.

**[PRS_SOMEIP_00925]** ⌈For the SOME/IP notification message the server has to do the following for payload and header:

- Construct the payload
- Set the Message ID based on the event the server wants to send
- Set the Length field to 8 bytes (for the part of the SOME/IP header after the length field) + length of the serialized payload
- Set the Client ID to `0x00`. Set the Session ID according to [PRS_SOMEIP_00932], [PRS_SOMEIP_00933], and [PRS_SOMEIP_00521]. In case of active Session Handling the Session ID shall be incremented upon each transmission.
- Set the Protocol Version according [PRS_SOMEIP_00052]
- Set the Interface Version according to the interface definition
- Set the Message Type to `NOTIFICATION` (i.e. `0x02`)
- Set the Return Code to `0x00`

⌋

**[PRS_SOMEIP_00926]** ⌈The payload of the notification message shall consist of the serialized data of the event.⌋

**[PRS_SOMEIP_00930]** ⌈When more than one subscribed client on the same ECU exists, the system shall handle the replication of notifications in order to save transmissions on the communication medium.⌋

This is especially important, when notifications are transported using multicast messages.

#### 5.2.4.1 Strategy for sending notifications

For different use cases different strategies for sending notifications are possible. The following examples are common:

- Cyclic update — send an updated value in a fixed interval (e.g. every 100 ms for safety relevant messages with Alive)
- Update on change — send an update as soon as a "value" changes (e.g. door open)
- Epsilon change — only send an update when the difference to the last value is greater than a certain epsilon. This concept may be adaptive, i.e. the prediction is based on a history; thus, only when the difference between prediction and current value is greater than epsilon an update is transmitted.

### 5.2.5 Fields

A field represents a status and has a valid value. The consumers subscribing for the field instantly after subscription get the field value as an initial event.

**[PRS_SOMEIP_00179]** ⌈A field shall be a combination of getter, setter and notification event.⌋

**[PRS_SOMEIP_00180]** ⌈A field without a setter and without a getter and without a notifier shall not exist. The field shall contain at least a getter, a setter, or a notifier.⌋

**[PRS_SOMEIP_00181]** ⌈The getter of a field shall be a request/response call that has an empty payload in the request message and the value of the field in the payload of the response message.⌋

**[PRS_SOMEIP_00182]** ⌈The setter of a field shall be a request/response call that has the desired value of the field in the payload of the request message and the value that was set to the field in the payload of the response message.⌋

**Note:**
If the value of the request payload was adapted (e.g. because it was out of limits) the adapted value will be transported in the response payload.

**[PRS_SOMEIP_00909]** ⌈The notifier shall send an event message that transports the value of the field to the client when the client subscribes to the field.⌋

**[PRS_SOMEIP_00183]** ⌈The notifier shall send an event message that transports the value of a field and shall follow the rules for events. Sending strategies include on change, on epsilon change, and cyclic sending.⌋

### 5.2.6 Error Handling

Error handling can be done in the application or the communication layer below. Therefore SOME/IP supports two different mechanisms:

- Return Codes in the Response Messages of methods
- Explicit Error Messages

Which one of both is used, depends on the configuration.

**[PRS_SOMEIP_00901]** ⌈Return Codes in the RESPONSE Messages (Message Type `0x80`) of methods shall be used to transport application errors and the response data of a method from the provider to the caller of a method. Message Type RESPONSE `0x80` shall be used in cases where no additional/extended error information (apart from the error code encoded in the Return Code field) needs to be propagated to the caller.⌋

**Note:**
Please be aware that return codes of the Request and Response methods are not treated as errors from the point of view of SOME/IP. This means that the message type is still `0x80` if a request/response method exits with a return code not equal to `0x00` (message type is still `0x80` if ApplicationError of AUTOSAR ClientServerOperation is different from E_OK).

**[PRS_SOMEIP_00902]** ⌈Explicit Error Messages shall be used to transport application errors and the response data or generic SOME/IP errors from the provider to the caller of a method.⌋

**[PRS_SOMEIP_00903]** ⌈If more detailed error information (apart from an error code encoded in the Return Code field) needs to be propagated to the caller, an ERROR message (Message type `0x81`) shall be used. The payload of the Error Message shall be filled with serialized error specific data, e.g. an exception string, or other error information. This additional error information shall be modeled in the interface description, or defined by the standard.⌋

This can be used to handle all different application errors that might occur in the server. In addition, problems with the communication medium or intermediate components (e.g. switches) may occur, which have to be handled e.g. by means of reliable transport.

All messages have a return code field in their header. (See chapter 5.1.1)

**[PRS_SOMEIP_00904]** ⌈Only responses (Response Messages (message type `0x80`) and Error Messages (message type 0x81) shall use the return code field to carry a return code to the request (Message Type `0x00`) they answer.⌋

**[PRS_SOMEIP_00905]** ⌈All other messages than `0x80` and `0x81` shall set this field to `0x00`.⌋

For message type see Chapter 5.1.1.7.

#### 5.2.6.1 Return Code

**[PRS_SOMEIP_00187]** ⌈The return code shall be UINT8.⌋

**[PRS_SOMEIP_00191] Return Codes** ⌈

| ID | Name | Description |
|---|---|---|
| 0x00 | E_OK | No error occurred. |
| 0x01 | E_NOT_OK | An unspecified error occurred. |
| 0x02 | E_UNKNOWN_SERVICE* | The requested Service ID is unknown. |
| 0x03 | E_UNKNOWN_METHOD* | The requested Method ID is unknown. Service ID is known. |
| 0x04 | E_NOT_READY | deprecated. |
| 0x05 | E_NOT_REACHABLE | deprecated. |
| 0x06 | E_TIMEOUT | deprecated. |
| 0x07 | E_WRONG_PROTOCOL_ VERSION | Version of SOME/IP protocol not supported. |
| 0x08 | E_WRONG_INTERFACE_ VERSION | Interface version mismatch. |
| 0x09 | E_MALFORMED_MESSAGE | Deserialization error, so that payload cannot be deserialized. |
| 0x0a | E_WRONG_MESSAGE_TYPE | An unexpected message type was received (e.g.received REQUEST for a method defined as REQUEST_NO_RETURN). |
| 0x0b - 0x0c | RESERVED | Reserved for generic SOME/IP errors. These errors will be specified in future versions of this document. |
| 0x0d | E_E2E | Not further specified E2E error. |
| 0x0e - 0x1f | RESERVED | Reserved for generic SOME/IP errors. These errors will be specified in future versions of this document. |
| 0x20 - 0x5E | RESERVED | Reserved for specific errors of services and methods. These errors are specified by the interface specification. |

⌋

Generation and handling of return codes shall be configurable.

**Note**:
`E_UNKNOWN_SERVICE` and `E_UNKNOWN_METHOD` are optional as they are implementation specific and depending on the implementation(classic platform or adaptive platform) they may or may not be returned as error codes.

**[PRS_SOMEIP_00539]** ⌈A SOME/IP error message (i.e. return code `0x01 - 0x1f`) shall not be answered with an error message.⌋

#### 5.2.6.2 Error Message

For more flexible error handling, SOME/IP allows a different message layout specific for Error Messages instead of using the message layout of Response Messages.

The recommended layout for the exception message is the following:

- Union of specific exceptions. At least a generic exception without fields needs to exist.
- Dynamic Length String for exception description.

Rationale: The union gives the flexibility to add new exceptions in the future in a type-safe manner. The string is used to transport human readable exception descriptions to ease testing and debugging.

**[PRS_SOMEIP_00188]** ⌈The receiver of a SOME/IP message shall not return an error message for events/notifications.⌋

**[PRS_SOMEIP_00189]** ⌈The receiver of a SOME/IP message shall not return an error message for fire&forget methods.⌋

**[PRS_SOMEIP_00537]** ⌈The receiver of a SOME/IP message shall only return an error message if the received message type is set to REQUEST.⌋

**Rationale:**
If the server maps the message ID to a NOTIFICATION or REQUEST_NO_RESPONSE type, the client that sends a message wrongly with type REQUEST would expect to receive a RESPONSE. RESPONSE with E_WRONG_MESSAGE_TYPE would make sense. On the other hand, if a client sends a message wrongly with type REQUEST_NO_RESPONSE, even if the server maps the message ID to a RESPONSE type, sending back an error RESPONSE is not helpful, as the client will simply ignore it.

**[PRS_SOMEIP_00190]** ⌈For Request/Response methods the error message shall copy over the fields of the SOME/IP header (i.e. Message ID, Request ID, and Interface Version) but not the payload. In addition Message Type and Return Code have to be set to the appropriate values.⌋

#### 5.2.6.3 Error Processing Overview

**[PRS_SOMEIP_00576]** ⌈Error handling shall be based on the message type received (e.g. only methods can be answered with a return code) and shall be checked in a defined order of [PRS_SOMEIP_00195].⌋

**[PRS_SOMEIP_00910]** ⌈For SOME/IP messages received over UDP, the following shall be checked:

- The UDP datagram size shall be at least 16 Bytes (minimum size of a SOME/IP message)
- The value of the length field shall be less than or equal to the remaining bytes in the UDP datagram payload

If one check fails, a malformed error shall be issued.⌋

**[PRS_SOMEIP_00195]** ⌈SOME/IP messages shall be checked by error processing. This does not include the application based error handling but just covers the error handling in messaging and RPC.⌋

An overview of the error processing is shown in Figure 5.12.

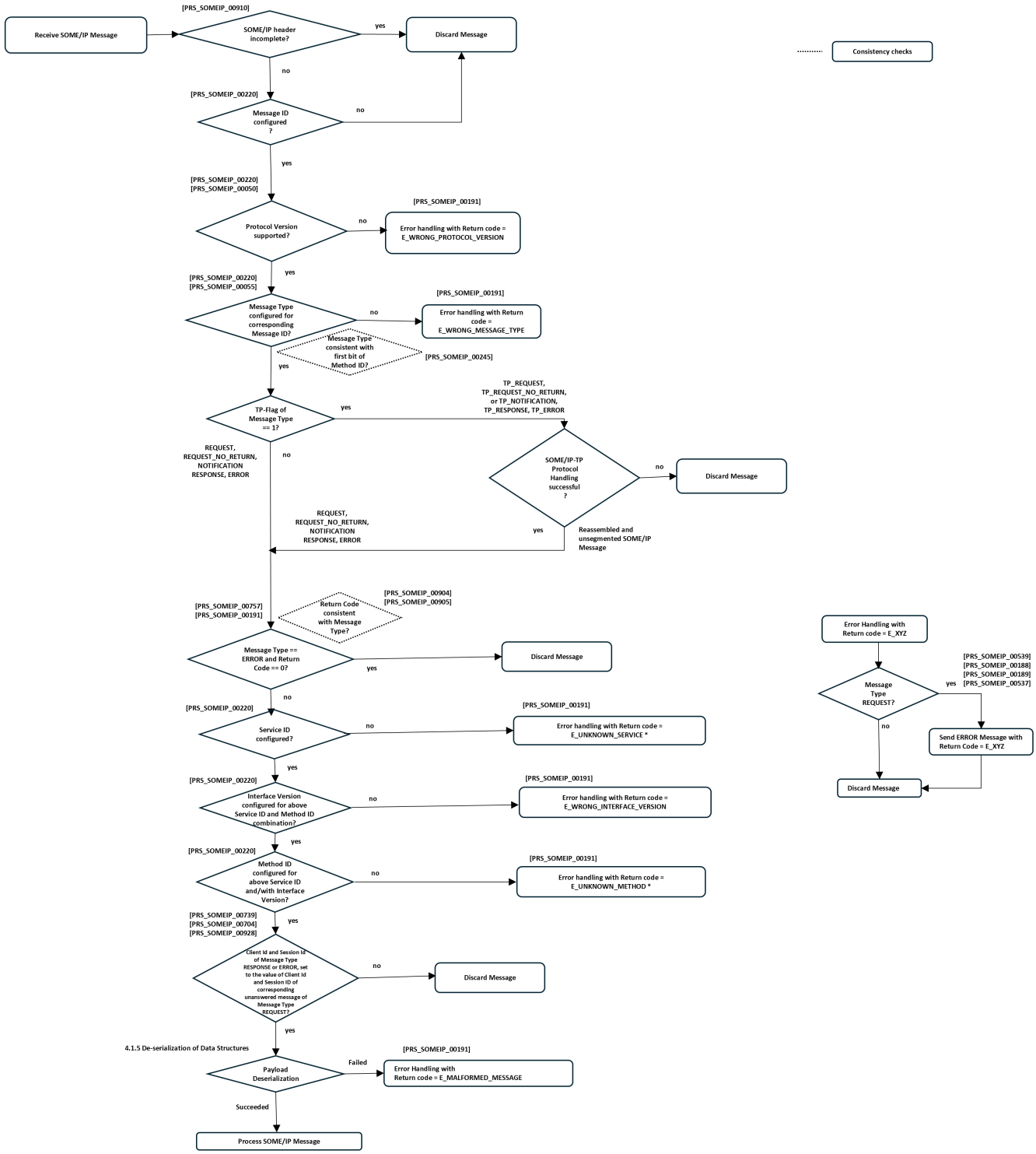


**Figure 5.12: Message Validation and Error Handling in SOME/IP**

**Note:** Certain sequence of the steps shown in this diagram can vary based on the specific implementation, e.g. Classic Platform splits the check for 'SOME/IP header incomplete?' between SocketAdaptor and SOME/IP Transformer.

**Note:** `E_UNKNOWN_SERVICE` and `E_UNKNOWN_METHOD` are optional as they are implementation specific and depending on the implementation(classic platform or adaptive platform) they may or may not be returned as error codes.

**[PRS_SOMEIP_00385] Conditions to discard received SOME/IP Message** ⌈Receiver shall discard SOME/IP message for the following scenarios of SOME/IP Header:

- Incomplete Message as per 'Length' Value
- 'Message ID' is invalid
- 'Message Type' is ERROR but Return Code is '0x00'
- 'Client ID' and 'Session ID' of messages with message type 'RESPONSE' or 'ERROR' are not set to the 'Client ID' and 'Session ID' of unanswered message with message type 'REQUEST'
- If the message type is not set to 'REQUEST' and below conditions are met:
  - 'Protocol Version' is not supported.
  - 'Service ID' is not configured
  - 'Interface Version' is not configured for received 'Service ID'
  - 'Method ID' is not configured for received 'Service ID' and 'Interface Version'
  - 'Message Type' is not configured for corresponding 'Message ID'(i.e,value of the 'Message Type' field in the received SOME/IP message does not match the 'Message Type' that is configured (according to the recipient's local configuration) for the value of the 'Message ID' field in the received SOME/IP message).
    For example,'Message ID' (according to local configuration) is configured for reception of method call RESPONSE(or ERROR) but 'Message Type' field in the actual received message is different from RESPONSE or ERROR.
    OR 'Message ID' (according to local configuration) is configured for reception of method REQUEST but 'Message Type' field in the actual received message is REQUEST_NO_RETURN.

⌋

#### 5.2.6.3.1 Handling of Communication Errors

When considering the transport of RPC messages different reliability semantics exist:

- Maybe — the message might reach the communication partner

- At least once — the message reaches the communication partner at least once
- Exactly once — the message reaches the communication partner exactly once

When using the above terms, in regard to Request/Response the term applies to both messages (i.e. request and response or error).

While different implementations may implement different approaches, SOME/IP currently achieves "maybe" reliability when using the UDP binding and "exactly once" reliability when using the TCP binding. Further error handling is left to the application.

For "maybe" reliability, only a single timeout is needed, when using request/response communication in combination of UDP as transport protocol. Figure 5.13 shows the state machines for "maybe" reliability. The client's SOME/IP implementation has to wait for the response for a specified timeout. If the timeout occurs SOME/IP shall signal E_TIMEOUT to the client application.

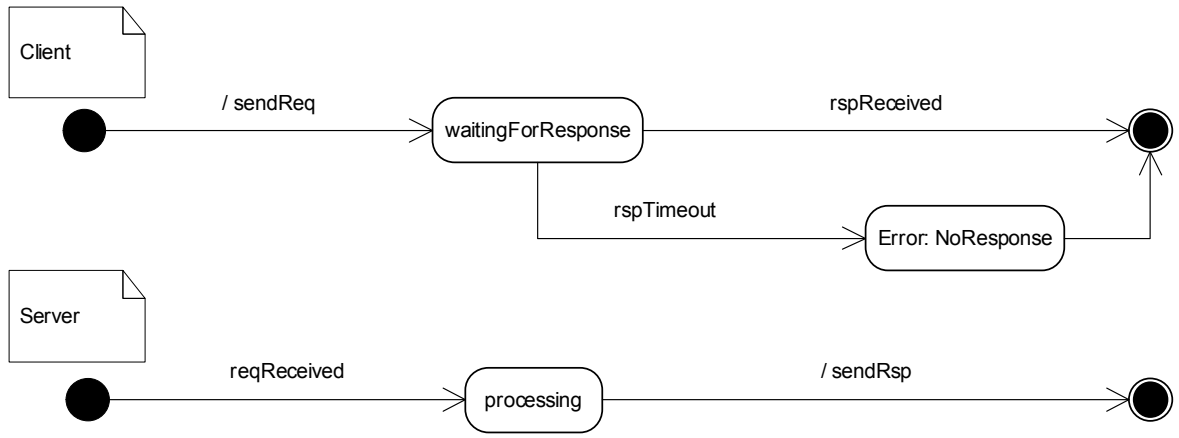


**Figure 5.13: State Machines for Reliability "Maybe"**

For "exactly once" reliability the TCP binding may be used, since TCP was defined to allow for reliable communication.

## 5.3 Compatibility Rules for Interface Version

The Interface Version identifies the Payload format. The Payload format is affected by

- the Service Interface specification
- the serialization configuration (e.g. usage of variable size arrays, size of length fields, padding, TLV, SOME/IP-TP).

**[PRS_SOMEIP_00937]** ⌈The Interface Version shall be increased for any of the following reasons:

- incompatible changes in the Payload format

- incompatible changes in the service behavior
- required by application design

⌋

Note: The Interface Version should not be increased for compatible changes in the Payload format. The rules in Chapter 7.3 define the compatibility of changes of the payload format.

# 6 Configuration Parameters

The following chapter summarizes the configuration parameters that are used in this specification.

**[PRS_SOMEIP_00378]** **SOMEIPTP_REASSEMBLY_TIMEOUT** ⌈SOME/IP Protocol shall provide a configurable timer SOMEIPTP_REASSEMBLY_TIMEOUT for the reception timeout time of SOME/IP-TP segments.⌋

| Parameter | Description | Parameter support in AP | Parameter support in CP |
|---|---|---|---|
| SOMEIPTP_REASSEMBLY_TIMEOUT ([PRS_SOMEIP_00378]) | Configurable timer for the reception timeout time between SOME/IP-TP segments | SomeipEventDeployment.segmentReceptionTimeoutTime [TPS_MANI_03328], SomeipMethodDeployment.segmentReceptionTimeoutTimeRequest [TPS_MANI_03329], SomeipMethodDeployment.segmentReceptionTimeoutTimeResponse [TPS_MANI_03330] | SomeipTpChannel.SomeipTpRxTimeoutTime see [ECUC_SomeIp-Tp_00023] |

**Table 6.1: Mapping Table - SOME/IP Protocol Parameters**

**[PRS_SOMEIP_00004] Configuration Parameters for SOME/IP (De)Serialization** ⌈The following table summarizes the configuration parameters that are used for SOME/IP (De)Serialization.⌋

| Parameter | Description | Allowed Range or Values | Default values(in case parameter is not explicitly configured) |
|---|---|---|---|
| ALIGNMENT | Defines the alignment requirement for the data element immediately following a variable length data element (if it is not the last element in the serialized data stream) in units of bytes. | 1,2,4,8,16,32 | 1 |
| BYTE_ORDER | Defines the byte order of the payload message. | mostSignificantByte-First, mostSignificantByteLast, opaque | mostSignificant-ByteFirst (BigEndian) |
| IS_ DYNAMIC_ LENGTH_ FIELD_SIZE | This parameter shall be used to determine the size of the length field based on wire type in the context of using the TLV encoding. If set to TRUE, the size of the length field defined in the data definition shall be ignored and the size of the length field shall be selected according to the wire type. | TRUE,FALSE | TRUE |
| SIZE_OF_ ARRAY_ LENGTH_ FIELD | Defines the size of the length field (in bytes) that will be put in front of an array (see Chapter 5.1.3.5) in the SOME/IP message. | 0, 1, 2, 4 for fixed length arrays (see [PRS_SOMEIP_ 00944]) where 0 indicates that there is no length field present. 1, 2, 4 for dynamic length arrays (see [PRS_SOMEIP_ 00001]) where 0 is not allowed since a dynamic length array has a mandatory length field. | 4 |

▽

△

| | | | |
|---|---|---|---|
| SIZE_OF_STRING_LENGTH_FIELD | Defines the size of the length field (in bytes) that will be put in front of a string (see Chapter 5.1.3.4) in the SOME/IP message. | 0, 1, 2, 4 for fixed length strings(see [PRS_SOMEIP_00760]) where 0 indicates that there is no length field present. 1, 2, 4 for dynamic length strings(see [PRS_SOMEIP_00002]) where 0 is not allowed since a dynamic length string has a mandatory length field. | 4 |
| SIZE_OF_STRUCT_LENGTH_FIELD | Defines the size of the length field (in bytes) that will be put in front of a struct (see Chapter 5.1.3.2) in the SOME/IP message. | 0,1,2,4 | 0 |
| SIZE_OF_UNION_LENGTH_FIELD | Defines the size of the length field (in bytes) that will be put in front of a union (see Chapter 5.1.3.8) in the SOME/IP message. | 0,1,2,4 | 4 |
| SIZE_OF_UNION_TYPE_SELECTOR_FIELD | Defines the size of the payload selector field (in bytes) that will be put in front of a union (see Chapter 5.1.3.8) in the SOME/IP message. | 1,2,4 | 4 |
| STRING_ENCODING | Defines the types of unicode encodings supported for a string (see Chapter 5.1.3.4) in the SOME/IP message. | UTF-8, UTF-16BE and UTF-16LE | UTF-8 |

**Table 6.2: SOME/IP Serialization Configuration Parameters**

# 7 Protocol usage and guidelines

## 7.1 Choosing the transport protocol

SOME/IP supports User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). While UDP is a very lean transport protocol supporting only the most important features (multiplexing and error detecting using a checksum), TCP adds additional features for achieving a reliable communication. TCP not only handles bit errors but also segmentation, loss, duplication, reordering, and network congestion.

Inside a vehicle many applications requires very short timeout to react quickly. These requirements are better met using UDP because the application itself can handle the unlikely event of errors. For example, in use cases with cyclic data it is often the best approach to just wait for the next data transmission instead of trying to repair the last one. The major disadvantage of UDP is that it does not handle segmentation. Hence, only being able to transport smaller chunks of data.

Guideline:

- Use TCP only if very large chunks of data need to be transported (> 1400 Bytes) and no hard latency requirements in the case of errors exists
- Use UDP if very hard latency requirements (<100ms) in case of errors is needed
- Use UDP together with SOME/IP-TP if very large chunks of data need to be transported (> 1400 Bytes) and hard latency requirements in the case of errors exists
- Try using external transport or transfer mechanisms (Network File System, APIX link, 1722, ...) when they are more suited for the use case. In this case SOME/IP can transport a file handle or a comparable identifier. This gives the designer additional freedom (e.g. in regard to caching).

The transport protocol used is specified by the interface specification on a per-message basis. Methods, Events, and Fields should commonly only use a single transport protocol.

## 7.2 Security Considerations for SOME/IP

**[PRS_SOMEIP_00946] Restricting Server Connection from Clients** ⌈A server may enforce communication policies to protect the server from malicious or unauthorized clients. I.e. the server may reject subscriptions to eventgroups, or method calls from unauthorized clients.⌋

**Note:** These policies are beyond the scope of this specification. Such policies can be based on the IP address of the client or any other means to identify the client.

**[PRS_SOMEIP_00947] Restricting Client Connection to Server** ⌈A client may enforce communication policies to protect the client from malicious servers. I.e. the client may deny communication to unauthorized servers.⌋

**Note:** These policies are beyond the scope of this specification. Such policies can be based on the IP address of the server or any other means to identify the client.

## 7.3 Compatibility of changes of the payload format

These table are based on the specification of the SOME/IP protocol. As a rule of thumb, interfaces are compatible if the receiver of data finds all expected information on the expected locations. For complex data types the rules are applied recursively.

For example, an entry in the table like the following:

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| Change Category | Detailed change | Transmitter | Receiver | Transmitter | Receiver | Transmitter | Receiver | Transmitter | Receiver |
| Add a struct member | Add a struct member to the end of the top-level struct MAXIMUM-MESSAGE-SIZE (MMS) | X | | X | | X | | X | X |

Should be interpreted as:

- If a Service interface "A1", which is serialized w/o length field and uses a struct as data type, and this struct is extended at the end with an further element, then the resulting Service interface A1' can be used in the following scenarios:
  - transmitter and receiver both updated to A1' -> compatible
  - transmitter updated to A1', receiver still A1 -> compatible
  - transmitter stays A1, receiver updated to A1' -> not compatible

The column "Transmitter" refers to the side transmitting a SOME/IP message, which can be the transmitter of an event (provider of the interface) or the method arguments used by the caller of a method (user of the interface) or the method result returned by the callee of a method (provider of the interface).
The column "Receiver" refers to the side receiving a SOME/IP message, which can be the receiver of an event (user of the interface) or the method arguments forwarded to the callee of a method (provider of the interface) or the method result forwarded to the caller of a message (user of the interface).
For fields, compatibility is only given if both the "Transmitter" and the "Receiver" columns are marked as compatible, because the same content will be transmitted by

the side defined by the provider of the interface (notification event, getter method) and the user of the interface (setter method).

The notation used in the table is as follows:

- `X` == compatible,
- `empty cell` == incompatible.
- `N/A` == not applicable (the field does not conceptually apply to this data type)

Compatibility of payload format changes by data type

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| **Change Category** | **Detailed change** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** |
| **Change: Basic Data Types** | | | | | | | | | |
| Change of basic data types | Change of data type to a larger or smaller one (e.g. uint8 to uint16) | | | | | | | | |
| | Change of data type to a syntactically different but semantically equal. (e.g. uint16 based enum/uint16) | X | X | X | X | X | X | X | X |
| | Change of data type to a semantically or syntactically different one (e.g. integer to struct, integer to float, string to string with different character size) | | | | | | | | |
| **Change: Structured DataTypes (structs)** | | | | | | | | | |
| Add a struct member | Add a struct member to the end of the top-level struct `MAXIMUM-MESSAGE-SIZE (MMS)` | X | | X | | X | | X | X |
| | Add a struct member to the end of a sub-struct `MAXIMUM-MESSAGE-SIZE (MMS)` | | | X | | X | | X | X |
| | Add a struct member **not** to the end of the struct `MAXIMUM-MESSAGE-SIZE (MMS)` | | | | | X | | X | X |
| Remove a struct member | Remove a struct member from the end of the top-level struct | | X | | X | | X | X | X |
| | Remove a struct member from the end of a sub-struct | | | | X | | X | X | X |
| | Remove a struct member **not** from the end of the struct | | | | | | X | X | X |
| Reorder struct members | | | | | | X | X | X | X |
| Change size of length field of a struct `VARIABLE-LENGTH-FIELD-SIZE` | | N/A | N/A | | | X | X | X | X |

△

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| Change Category | Detailed change | Transmitter | Receiver | Transmitter | Receiver | Transmitter | Receiver | Transmitter | Receiver |
| Change data ID of a struct member | | N/A | N/A | N/A | N/A | | | | |
| Reuse data ID of a previously removed struct member | | N/A | N/A | N/A | N/A | | | | |
| **Change: Strings** | | | | | | | | | |
| Change the length of a fixed size string | Increase the length of a fixed size string ZERO-TERMINATED | | | | X | | X | | X |
| | Decrease the length of a fixed size string ZERO-TERMINATED | | | X | | X | | X | |
| Change the maximum length of a variable size string | Increase maximum length of a variable size string | N/A | N/A | | X | | X | | X |
| | Decrease maximum length of a variable size string | N/A | N/A | X | | X | | X | |
| Change size of length field of a string VARIABLE-LENGTH-FIELD-SIZE | | N/A | N/A | | | X | X | X | X |
| **Change: Arrays** | | | | | | | | | |
| Change the length of a fixed size array | Increase the length of a fixed size array | | | X | | X | | X | |
| | Decrease the length of a fixed size array | | | | X | | X | | X |
| Change maximum length of a variable size array | | N/A | N/A | X | X | X | X | X | X |
| Change number of dimensions of an array | | | | | | | | | |
| Change size of length field of an array VARIABLE-LENGTH-FIELD-SIZE | | N/A | N/A | | | X | X | X | X |
| **Change: Enumeration** | | | | | | | | | |
| Add/Remove enumeration values | Add new enumeration values ENUM-REMARK | | X | | X | | X | | X |
| | Remove enumeration values ENUM-REMARK | X | | X | | X | | X | |
| Change an existing enumeration value | Change existing enumeration values ENUM-REMARK | | | | | | | | |
| **Change: Union** | | | | | | | | | |
| Add a union member | Add a new union member with previously unused type selector | | X | | X | | X | | X |
| Remove a union member | Remove a union member with highest type selector | X | | X | | X | | X | |
| | Remove a union member with non-highest type selector | | | | | | | | |

▽

△

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| **Change Category** | **Detailed change** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** |
| Change a type selector | Redefine the type selector of non-highest union member | | | | | | | | |
| | Change size of a union type selector | | | | | | | | |
| Change size of length field of a union VARIABLE-LENGTH-FIELD-SIZE | | N/A | N/A | | | X | X | X | X |

## Compatibility of payload format changes by signature of the method

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| **Change Category** | **Detailed change** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** |
| **Change: signature of the method** | | | | | | | | | |
| Add an argument of a method request | Add an argument to the end of the argument list of a method request MAXIMUM-MESSAGE-SIZE (MMS), OPTIONAL-METHOD-ARGS-REMARK | X | | X | | X | | X | X |
| | Add an argument **not** to the end of the argument list of a method request MAXIMUM-MESSAGE-SIZE (MMS), OPTIONAL-METHOD-ARGS-REMARK | | | | | X | | X | X |
| Remove an argument of a method request | Remove an argument from the end of the argument list of a method request OPTIONAL-METHOD-ARGS-REMARK | | X | | X | | X | X | X |
| | Remove an argument **not** from the end of the argument list of a method request OPTIONAL-METHOD-ARGS-REMARK | | | | | | X | X | X |
| Add an argument of a method response | Add an argument to the end of the argument list of a method response MAXIMUM-MESSAGE-SIZE (MMS), OPTIONAL-METHOD-ARGS-REMARK | X | | X | | X | | X | X |
| | Add an argument **not** to the end of the argument list of a method response MAXIMUM-MESSAGE-SIZE (MMS), OPTIONAL-METHOD-ARGS-REMARK | | | | | X | | X | X |

▽

△

| Change of Interface | | Classes of Protocol / Serialization Capabilities | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Serialization without length fields | | Serialization with length fields | | Serialization with TLV | | Serialization with TLV and optional members | |
| **Change Category** | **Detailed change** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** | **Transmitter** | **Receiver** |
| Remove an argument of a method response | Remove an argument from the end of the argument list of a method response OPTIONAL-METHOD-ARGS-REMARK | | X | | X | | X | X | X |
| | Remove an argument **not** from the end of the argument list of a method response OPTIONAL-METHOD-ARGS-REMARK | | | | | | X | X | X |
| Reorder arguments of methods | | | | | | X | X | X | X |
| Change optionality of arguments OPTIONAL-METHOD-ARGS-REMARK | | N/A | N/A | N/A | N/A | N/A | N/A | | |
| Change data ID of arguments of methods | | N/A | N/A | N/A | N/A | | | | |
| Reuse data ID of previously removed arguments | | N/A | N/A | N/A | N/A | | | | |
| Add/Remove the return code of a method | Add return codes of a method | | X | | X | | X | | X |
| | Remove return codes of a method | X | | X | | X | | X | |
| Change the return code of a method | Change an existing return code of a method | | | | | | | | |

## Compatibility of payload format changes by service interface definition

| Change of Interface | | Classes of Protocol / Serialization Capabilities | |
|---|---|---|---|
| **Change Category** | **Detailed change** | **Server** | **Client** |
| **Change: service interface definition** | | | |
| Add/Remove a method of a service interface | Add a new method to a service interface | X | |
| | Remove a method from a service interface | | X |
| Add/Remove a setter/getter/notifier of a field | Add setter or getter to a field | X | |
| | Add notifier to a field | X | |
| | Remove setter or getter from a field | | X |
| | Remove notifier from a field | | X |
| Add/Remove an event of an eventgroup | Add an event to eventgroup | X | |
| | Remove an event from eventgroup | | X |
| Change the name of interfaces | Change of the name of a service interface, method, field, eventgroup or event | | |
| Change Method ID | | | |

## Compatibility of payload format changes by serialization configuration

| Change of Interface | | Classes of Protocol / Serialization Capabilities | |
|---|---|---|---|
| **Change Category** | **Detailed change** | **Transmitter** | **Receiver** |
| **Change: serialization configuration** | | | |
| Change the alignment configuration | Change the alignment sizes of data behind variable length data elements | | |
| Change the byte order of arguments | | | |
| Change the byte order of Event data | | | |
| Change to/from static/dynamic TLV | | | |
| Change to/from TLV/non-TLV | | | |
| Change string encoding to/from UTF-8/UTF-16 | | | |

The following legend to the table applies:

ENUM-REMARK: The "Receiver" of an Enumeration needs to know all values it can receive, while the "Transmitter" does not have to send all values the "Receiver" knows.

MAXIMUM-MESSAGE-SIZE (MMS): The compatibility in Classic Platform is limited by the maximum configured size of message. This means that if the message size is configured larger than necessary, additional text could be added and, independently of the message size on tag, can be replaced by a new one.

OPTIONAL-METHOD-ARGS-REMARK: See second note of [PRS_SOMEIP_00201].

ZERO-TERMINATED: Strings are terminated with a "\0"-character(see [PRS_SOMEIP_00373], [PRS_SOMEIP_00911], [PRS_SOMEIP_00912] and [PRS_SOMEIP_00913])

VARIABLE-LENGTH-FIELD-SIZE: The size of the length field could be selected according to the wire type 5, 6 or 7.(see [PRS_SOMEIP_00205], [PRS_SOMEIP_00206].)

# A Change history of AUTOSAR traceable items

Please note that the lists in this chapter also include traceable items that have been removed from the specification in a later version. These items do not appear as hyperlinks in the document.

## A.1 Traceable item history of this document according to AUTOSAR Release R26-11

### A.1.1 Added Specification Items in R26-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00088] | Allowed characters in string |
| [PRS_SOMEIP_00949] | Server Response Addressing |
| [PRS_SOMEIP_00950] | Concurrent Requests Handling |
| [PRS_SOMEIP_00951] | Padding requirement for strings |

**Table A.1: Added Specification Items in R26-11**

### A.1.2 Changed Specification Items in R26-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00055] | Message Type field |
| [PRS_SOMEIP_00126] | |
| [PRS_SOMEIP_00191] | Return Codes |
| [PRS_SOMEIP_00370] | |
| [PRS_SOMEIP_00377] | |
| [PRS_SOMEIP_00532] | |
| [PRS_SOMEIP_00611] | |
| [PRS_SOMEIP_00702] | Client ID uniqueness within a machine |



△

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00911] | Handling of Fixed-Length Strings exceeding defined size |
| [PRS_SOMEIP_00912] | Handling of Fixed-Length Strings with reduced size |

**Table A.2: Changed Specification Items in R26-11**

### A.1.3 Deleted Specification Items in R26-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00758] | Compatibility of changes of the payload format |
| [PRS_SOMEIP_00938] | |

**Table A.3: Deleted Specification Items in R26-11**

### A.1.4 Added Constraints in R26-11

none

### A.1.5 Changed Constraints in R26-11

none

### A.1.6 Deleted Constraints in R26-11

none

## A.2 Traceable item history of this document according to AUTOSAR Release R25-11

### A.2.1 Added Specification Items in R25-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00001] | Length field for Dynamic length arrays |
| [PRS_SOMEIP_00002] | Length field for Dynamic length strings |
| [PRS_SOMEIP_00003] | Selection of to-be-used wire type and length of the Length field |
| [PRS_SOMEIP_00004] | Configuration Parameters for SOME/IP (De)Serialization |
| [PRS_SOMEIP_00207] | Behaviour if a required element is missing in the received SOME/IP message |
| [PRS_SOMEIP_00209] | Behaviour if a required member/argument is missing in the received SOME/IP message |
| [PRS_SOMEIP_00383] | Behaviour if a required member/argument is found with invalid wire type in the received SOME/IP message |
| [PRS_SOMEIP_00384] | Behaviour if a required member/argument is found multiple times in the received SOME/IP message |
| [PRS_SOMEIP_00385] | Conditions to discard received SOME/IP Message |
| [PRS_SOMEIP_00939] | Session Handling for `REQUEST_NO_RETURN` Messages |

**Table A.4: Added Specification Items in R25-11**

### A.2.2 Changed Specification Items in R25-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00050] | |
| [PRS_SOMEIP_00079] | |
| [PRS_SOMEIP_00121] | |
| [PRS_SOMEIP_00127] | |
| [PRS_SOMEIP_00191] | Return Codes |
| [PRS_SOMEIP_00206] | |

▽

△

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00225] | |
| [PRS_SOMEIP_00227] | |
| [PRS_SOMEIP_00369] | |
| [PRS_SOMEIP_00372] | |
| [PRS_SOMEIP_00613] | |
| [PRS_SOMEIP_00701] | |
| [PRS_SOMEIP_00760] | |
| [PRS_SOMEIP_00900] | |
| [PRS_SOMEIP_00944] | |

**Table A.5: Changed Specification Items in R25-11**

### A.2.3 Deleted Specification Items in R25-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00093] | |
| [PRS_SOMEIP_00094] | |
| [PRS_SOMEIP_00107] | |
| [PRS_SOMEIP_00380] | Behaviour if a required member/argument is missing in the received SOME/IP message |
| [PRS_SOMEIP_00381] | Behaviour if a required element is missing in the received SOME/IP message |
| [PRS_SOMEIP_00614] | |
| [PRS_SOMEIP_00945] | |

**Table A.6: Deleted Specification Items in R25-11**

### A.2.4 Added Constraints in R25-11

| Number | Heading |
|---|---|
| [constr_00001] | Reordering limitation for a SOME/IP message segment |

**Table A.7: Added Constraints in R25-11**

### A.2.5 Changed Constraints in R25-11

none

### A.2.6 Deleted Constraints in R25-11

none

## A.3 Traceable item history of this document according to AUTOSAR Release R24-11

### A.3.1 Added Specification Items in R24-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00378] | `SOMEIPTP_REASSEMBLY_TIMEOUT` |
| [PRS_SOMEIP_00379] | . |
| [PRS_SOMEIP_00380] | Behaviour if a required member/argument is missing in the received SOME/IP message |
| [PRS_SOMEIP_00381] | Behaviour if a required element is missing in the received SOME/IP message |
| [PRS_SOMEIP_00382] | SOME/IP payload size using UDP |

**Table A.8: Added Specification Items in R24-11**

### A.3.2 Changed Specification Items in R24-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00119] | Union (length field, type selector field and payload) |
| [PRS_SOMEIP_00706] | |
| [PRS_SOMEIP_00730] | Maximum segement length |
| [PRS_SOMEIP_00758] | Compatibility of changes of the payload format |
| [PRS_SOMEIP_00900] | |
| [PRS_SOMEIP_00901] | |
| [PRS_SOMEIP_00903] | |
| [PRS_SOMEIP_00907] | |
| [PRS_SOMEIP_00942] | |

**Table A.9: Changed Specification Items in R24-11**

### A.3.3 Deleted Specification Items in R24-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00218] | |
| [PRS_SOMEIP_00908] | |
| [PRS_SOMEIP_00918] | |

**Table A.10: Deleted Specification Items in R24-11**

## A.4 Traceable item history of this document according to AUTOSAR Release R23-11

### A.4.1 Added Specification Items in R23-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00755] | Message ID header field |
| [PRS_SOMEIP_00757] | Return Codes |
| [PRS_SOMEIP_00758] | Compatibility of changes of the payload format |
| [PRS_SOMEIP_00759] | |
| [PRS_SOMEIP_00760] | |

**Table A.11: Added Specification Items in R23-11**

### A.4.2 Changed Specification Items in R23-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00046] | Request ID |
| [PRS_SOMEIP_00107] | |
| [PRS_SOMEIP_00163] | |
| [PRS_SOMEIP_00183] | |
| [PRS_SOMEIP_00205] | Wire Type and type of following data |
| [PRS_SOMEIP_00369] | |
| [PRS_SOMEIP_00376] | |

**Table A.12: Changed Specification Items in R23-11**

### A.4.3 Deleted Specification Items in R23-11

none

## A.5 Traceable item history of this document according to AUTOSAR Release R22-11

### A.5.1 Added Specification Items in R22-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00245] | |

**Table A.13: Added Specification Items in R22-11**

### A.5.2 Changed Specification Items in R22-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_00043] | |
| [PRS_SOMEIP_00101] | |
| [PRS_SOMEIP_00137] | |
| [PRS_SOMEIP_00241] | |
| [PRS_SOMEIP_00300] | |
| [PRS_SOMEIP_00365] | |
| [PRS_SOMEIP_00366] | |
| [PRS_SOMEIP_00374] | |
| [PRS_SOMEIP_00730] | Maximum segement length |
| [PRS_SOMEIP_00739] | |
| [PRS_SOMEIP_00922] | |
| [PRS_SOMEIP_00942] | |

**Table A.14: Changed Specification Items in R22-11**

### A.5.3 Deleted Specification Items in R22-11

| Number | Heading |
|---|---|
| [PRS_SOMEIP_ 00038] | |
| [PRS_SOMEIP_ 00040] | |

**Table A.15: Deleted Specification Items in R22-11**

[1] Specification of Service Discovery
AUTOSAR_CP_SWS_ServiceDiscovery

[2] Glossary
AUTOSAR_FO_TR_Glossary