Document Title	SOME/IP Protocol Specification
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	696

Document Status	published
Part of AUTOSAR Standard	Foundation
Part of Standard Release	R20-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2020-11-30	R20-11	AUTOSAR Release Management	• Removed Draft Status from TLV Requirements • Fixed discrepancies between SWS and PRS • Clarified usage of length field • Restricted alignment of variable length arrays to 8, 16, 32, 64, 128 or 256 Bits • Editorial Changes
2019-11-28	R19-11	AUTOSAR Release Management	• Added – Support for unit64 / sint64 – Error-Codes for E2E-Protection • Clarify – Serialization of fixed length array data – Support for Data Accumulation feature – Contradicting requirements • Introduce implementsLegacyStringSerialization tag (as successor of implementsSOMEIPStringHandling) • Editorial Changes • 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 • 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

Table of Contents

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

The basic motivation to specify "Yet another RPC-Mechanism" instead of using an existing infrastructure/technology is the goal to have a technology that:

- Fulfills the hard requirements regarding resource consumption in an embedded world
- Is compatible through as many use-cases and communication partners as possible
- compatible with AUTOSAR at least on the wire-format level; i.e. can communicate with PDUs AUTOSAR can receive and send without modification to the AUTOSAR standard. The mappings within AUTOSAR shall be chosen according to the SOME/IP specification.
- 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, 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 signals 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) cannot process. To indicate this interoperability issue

[PRS_SOMEIP_00220] requires to increase the major interface version of the respective serialized data.

1.2.2 Limitations

This document gives a holistic overview over SOME/IP but doesn't state any requirements towards any implementation of BSW modules.

Please be aware that not all parts of SOME/IP may be implemented in AUTOSAR.

1.3 Dependencies

There are no dependencides 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

2 Protocol Requirements

2.1 Requirements Traceability

Feature	Description	Satisfied by
[RS_SOMEIP_00002]	SOME/IP protocol shall provide service-based communication	[PRS_SOMEIP_00043] [PRS_SOMEIP_00703] [PRS_SOMEIP_00909]
[RS_SOMEIP_00003]	SOME/IP protocol shall provide support of multiple versions of a service interface	[PRS_SOMEIP_00053] [PRS_SOMEIP_00937] [PRS_SOMEIP_00938]
[RS_SOMEIP_00004]	SOME/IP protocol shall support event communication	[PRS_SOMEIP_00925] [PRS_SOMEIP_00926]
[RS_SOMEIP_00005]	SOME/IP protocol shall support different strategies for event communication	[PRS_SOMEIP_00183]
[RS_SOMEIP_00006]	SOME/IP protocol shall support uni-directional RPC communication	[PRS_SOMEIP_00171] [PRS_SOMEIP_00924]
[RS_SOMEIP_00007]	SOME/IP protocol shall support bi-directional RPC communication	[PRS_SOMEIP_00920] [PRS_SOMEIP_00921] [PRS_SOMEIP_00922] [PRS_SOMEIP_00923] [PRS_SOMEIP_00927] [PRS_SOMEIP_00928]
[RS_SOMEIP_00008]	SOME/IP protocol shall support error handling of RPC communication	[PRS_SOMEIP_00055] [PRS_SOMEIP_00058] [PRS_SOMEIP_00187] [PRS_SOMEIP_00188] [PRS_SOMEIP_00189] [PRS_SOMEIP_00190] [PRS_SOMEIP_00191] [PRS_SOMEIP_00195] [PRS_SOMEIP_00537] [PRS_SOMEIP_00539] [PRS_SOMEIP_00576] [PRS_SOMEIP_00614] [PRS_SOMEIP_00701] [PRS_SOMEIP_00901] [PRS_SOMEIP_00902] [PRS_SOMEIP_00903] [PRS_SOMEIP_00904] [PRS_SOMEIP_00905] [PRS_SOMEIP_00910]
[RS_SOMEIP_00009]	SOME/IP protocol shall support field communication	[PRS_SOMEIP_00179] [PRS_SOMEIP_00180] [PRS_SOMEIP_00181] [PRS_SOMEIP_00182] [PRS_SOMEIP_00183] [PRS_SOMEIP_00909]

[RS_SOMEIP_00010]	SOME/IP protocol shall support different transport protocols underneath	[PRS_SOMEIP_00137] [PRS_SOMEIP_00139] [PRS_SOMEIP_00140] [PRS_SOMEIP_00141] [PRS_SOMEIP_00142] [PRS_SOMEIP_00154] [PRS_SOMEIP_00160] [PRS_SOMEIP_00535] [PRS_SOMEIP_00706] [PRS_SOMEIP_00707] [PRS_SOMEIP_00708] [PRS_SOMEIP_00709] [PRS_SOMEIP_00710] [PRS_SOMEIP_00711] [PRS_SOMEIP_00720] [PRS_SOMEIP_00721] [PRS_SOMEIP_00722] [PRS_SOMEIP_00723] [PRS_SOMEIP_00724] [PRS_SOMEIP_00725] [PRS_SOMEIP_00726] [PRS_SOMEIP_00727] [PRS_SOMEIP_00728] [PRS_SOMEIP_00729] [PRS_SOMEIP_00730] [PRS_SOMEIP_00731] [PRS_SOMEIP_00732] [PRS_SOMEIP_00733] [PRS_SOMEIP_00734] [PRS_SOMEIP_00735] [PRS_SOMEIP_00736] [PRS_SOMEIP_00738] [PRS_SOMEIP_00740] [PRS_SOMEIP_00741] [PRS_SOMEIP_00742] [PRS_SOMEIP_00743] [PRS_SOMEIP_00744] [PRS_SOMEIP_00745] [PRS_SOMEIP_00746] [PRS_SOMEIP_00747] [PRS_SOMEIP_00749] [PRS_SOMEIP_00750] [PRS_SOMEIP_00751] [PRS_SOMEIP_00752] [PRS_SOMEIP_00753] [PRS_SOMEIP_00754] [PRS_SOMEIP_00940] [PRS_SOMEIP_00942] [PRS_SOMEIP_00943]
[RS_SOMEIP_00011]	SOME/IP protocol shall support messages of different lengths	[PRS_SOMEIP_00722]

[RS_SOMEIP_00012]	SOME/IP protocol shall support session handling	[PRS_SOMEIP_00521] [PRS_SOMEIP_00533] [PRS_SOMEIP_00720] [PRS_SOMEIP_00721] [PRS_SOMEIP_00739] [PRS_SOMEIP_00935] [PRS_SOMEIP_00936]
[RS_SOMEIP_00014]	SOME/IP protocol shall support handling of protocol errors on receiver side	[PRS_SOMEIP_00195] [PRS_SOMEIP_00576] [PRS_SOMEIP_00614] [PRS_SOMEIP_00910]
[RS_SOMEIP_00015]	SOME/IP protocol shall support multiple instances of a service	[PRS_SOMEIP_00138] [PRS_SOMEIP_00162] [PRS_SOMEIP_00163]
[RS_SOMEIP_00016]	SOME/IP protocol shall support combining multiple RPC methods, events and fields in one service	[PRS_SOMEIP_00040] [PRS_SOMEIP_00366]
[RS_SOMEIP_00017]	SOME/IP protocol shall support grouping events into eventgroups	[PRS_SOMEIP_00365] [PRS_SOMEIP_00366]
[RS_SOMEIP_00018]	SOME/IP protocol shall support grouping fields in eventgroups	[PRS_SOMEIP_00366]
[RS_SOMEIP_00021]	SOME/IP protocol shall identify RPC methods of services using unique identifiers	[PRS_SOMEIP_00034]
[RS_SOMEIP_00022]	SOME/IP protocol shall identify events of services using unique identifiers	[PRS_SOMEIP_00034]
[RS_SOMEIP_00023]	SOME/IP protocol shall identify event groups of services using unique identifiers	[PRS_SOMEIP_00034]
[RS_SOMEIP_00024]	SOME/IP protocol shall define reserved identifiers	[PRS_SOMEIP_00191] [PRS_SOMEIP_00907]
[RS_SOMEIP_00025]	SOME/IP protocol shall support the identification of callers of an RPC using unique identifiers	[PRS_SOMEIP_00043] [PRS_SOMEIP_00044] [PRS_SOMEIP_00532] [PRS_SOMEIP_00702] [PRS_SOMEIP_00703]
[RS_SOMEIP_00026]	SOME/IP protocol shall define the endianness of header and payload	[PRS_SOMEIP_00368] [PRS_SOMEIP_00369]

[RS_SOMEIP_00027]	SOME/IP protocol shall define the header layout of messages	[PRS_SOMEIP_00030] [PRS_SOMEIP_00031] [PRS_SOMEIP_00034] [PRS_SOMEIP_00038] [PRS_SOMEIP_00040] [PRS_SOMEIP_00042] [PRS_SOMEIP_00043] [PRS_SOMEIP_00046] [PRS_SOMEIP_00050] [PRS_SOMEIP_00051] [PRS_SOMEIP_00052] [PRS_SOMEIP_00053] [PRS_SOMEIP_00055] [PRS_SOMEIP_00058] [PRS_SOMEIP_00141] [PRS_SOMEIP_00365] [PRS_SOMEIP_00366] [PRS_SOMEIP_00367] [PRS_SOMEIP_00521] [PRS_SOMEIP_00532] [PRS_SOMEIP_00533] [PRS_SOMEIP_00701] [PRS_SOMEIP_00702] [PRS_SOMEIP_00703] [PRS_SOMEIP_00704] [PRS_SOMEIP_00723] [PRS_SOMEIP_00724] [PRS_SOMEIP_00725] [PRS_SOMEIP_00726] [PRS_SOMEIP_00727] [PRS_SOMEIP_00728] [PRS_SOMEIP_00739] [PRS_SOMEIP_00931] [PRS_SOMEIP_00932] [PRS_SOMEIP_00933] [PRS_SOMEIP_00934] [PRS_SOMEIP_00935] [PRS_SOMEIP_00936] [PRS_SOMEIP_00940] [PRS_SOMEIP_00941]

[RS_SOMEIP_00028]	SOME/IP protocol shall specify the serialization algorithm for data	[PRS_SOMEIP_00101] [PRS_SOMEIP_00130] [PRS_SOMEIP_00210] [PRS_SOMEIP_00211] [PRS_SOMEIP_00212] [PRS_SOMEIP_00213] [PRS_SOMEIP_00214] [PRS_SOMEIP_00216] [PRS_SOMEIP_00220] [PRS_SOMEIP_00569] [PRS_SOMEIP_00611] [PRS_SOMEIP_00612] [PRS_SOMEIP_00613] [PRS_SOMEIP_00712] [PRS_SOMEIP_00921] [PRS_SOMEIP_00923]
[RS_SOMEIP_00029]	SOME/IP protocol shall specify how data in the payload are aligned	[PRS_SOMEIP_00222] [PRS_SOMEIP_00569] [PRS_SOMEIP_00611] [PRS_SOMEIP_00612] [PRS_SOMEIP_00613] [PRS_SOMEIP_00730]
[RS_SOMEIP_00030]	SOME/IP protocol shall support transporting integer data types	[PRS_SOMEIP_00065] [PRS_SOMEIP_00300] [PRS_SOMEIP_00615] [PRS_SOMEIP_00705]
[RS_SOMEIP_00031]	SOME/IP protocol shall support transporting boolean data type	[PRS_SOMEIP_00065] [PRS_SOMEIP_00615]
[RS_SOMEIP_00032]	SOME/IP protocol shall support transporting float data types	[PRS_SOMEIP_00065] [PRS_SOMEIP_00615]
[RS_SOMEIP_00033]	SOME/IP protocol shall support transporting structured data types	[PRS_SOMEIP_00077] [PRS_SOMEIP_00079] [PRS_SOMEIP_00300] [PRS_SOMEIP_00370] [PRS_SOMEIP_00371] [PRS_SOMEIP_00705] [PRS_SOMEIP_00712] [PRS_SOMEIP_00900]
[RS_SOMEIP_00034]	SOME/IP protocol shall support transporting union data types	[PRS_SOMEIP_00118] [PRS_SOMEIP_00119] [PRS_SOMEIP_00121] [PRS_SOMEIP_00122] [PRS_SOMEIP_00123] [PRS_SOMEIP_00126] [PRS_SOMEIP_00127] [PRS_SOMEIP_00129] [PRS_SOMEIP_00130] [PRS_SOMEIP_00906] [PRS_SOMEIP_00907] [PRS_SOMEIP_00908] [PRS_SOMEIP_00915] [PRS_SOMEIP_00916]

[RS_SOMEIP_00035]	SOME/IP protocol shall support transporting one-dimensional and multi-dimensional array data types	[PRS_SOMEIP_00099] [PRS_SOMEIP_00101]
[RS_SOMEIP_00036]	SOME/IP protocol shall support transporting array data types with a fixed length	[PRS_SOMEIP_00099] [PRS_SOMEIP_00101] [PRS_SOMEIP_00917] [PRS_SOMEIP_00918] [PRS_SOMEIP_00944]
[RS_SOMEIP_00037]	SOME/IP protocol shall support transporting array data types with flexible length	[PRS_SOMEIP_00107] [PRS_SOMEIP_00114] [PRS_SOMEIP_00375] [PRS_SOMEIP_00376] [PRS_SOMEIP_00377] [PRS_SOMEIP_00919] [PRS_SOMEIP_00945]
[RS_SOMEIP_00038]	SOME/IP protocol shall support transporting string types with a fixed length	[PRS_SOMEIP_00084] [PRS_SOMEIP_00085] [PRS_SOMEIP_00086] [PRS_SOMEIP_00087] [PRS_SOMEIP_00372] [PRS_SOMEIP_00373] [PRS_SOMEIP_00374] [PRS_SOMEIP_00911] [PRS_SOMEIP_00912] [PRS_SOMEIP_00913]
[RS_SOMEIP_00039]	SOME/IP protocol shall support transporting string data types with flexible length	[PRS_SOMEIP_00089] [PRS_SOMEIP_00090] [PRS_SOMEIP_00091] [PRS_SOMEIP_00092] [PRS_SOMEIP_00093] [PRS_SOMEIP_00094] [PRS_SOMEIP_00095] [PRS_SOMEIP_00914]
[RS_SOMEIP_00040]	SOME/IP protocol shall support providing the length of a serialized data element in the payload	[PRS_SOMEIP_00042] [PRS_SOMEIP_00079] [PRS_SOMEIP_00094] [PRS_SOMEIP_00208] [PRS_SOMEIP_00221] [PRS_SOMEIP_00370] [PRS_SOMEIP_00945]
[RS_SOMEIP_00041]	SOME/IP protocol shall provide support of multiple versions of the protocol	[PRS_SOMEIP_00050] [PRS_SOMEIP_00051] [PRS_SOMEIP_00052]
[RS_SOMEIP_00042]	SOME/IP protocol shall support unicast and multicast based event communication	[PRS_SOMEIP_00930]

[RS_SOMEIP_00050]	SOME/IP protocol shall support serialization of extensible data structs	[PRS_SOMEIP_00201] [PRS_SOMEIP_00202] [PRS_SOMEIP_00203] [PRS_SOMEIP_00204] [PRS_SOMEIP_00205] [PRS_SOMEIP_00206] [PRS_SOMEIP_00208] [PRS_SOMEIP_00210] [PRS_SOMEIP_00211] [PRS_SOMEIP_00212] [PRS_SOMEIP_00213] [PRS_SOMEIP_00214] [PRS_SOMEIP_00216] [PRS_SOMEIP_00217] [PRS_SOMEIP_00218] [PRS_SOMEIP_00220] [PRS_SOMEIP_00221] [PRS_SOMEIP_00222] [PRS_SOMEIP_00223] [PRS_SOMEIP_00224] [PRS_SOMEIP_00225] [PRS_SOMEIP_00226] [PRS_SOMEIP_00227] [PRS_SOMEIP_00228] [PRS_SOMEIP_00229] [PRS_SOMEIP_00230] [PRS_SOMEIP_00231] [PRS_SOMEIP_00241] [PRS_SOMEIP_00242] [PRS_SOMEIP_00243] [PRS_SOMEIP_00244]

[RS_SOMEIP_00051]	SOME/IP protocol shall provide support for segmented transmission of large data	[PRS_SOMEIP_00367] [PRS_SOMEIP_00729] [PRS_SOMEIP_00730] [PRS_SOMEIP_00731] [PRS_SOMEIP_00732] [PRS_SOMEIP_00733] [PRS_SOMEIP_00734] [PRS_SOMEIP_00735] [PRS_SOMEIP_00736] [PRS_SOMEIP_00738] [PRS_SOMEIP_00740] [PRS_SOMEIP_00741] [PRS_SOMEIP_00742] [PRS_SOMEIP_00743] [PRS_SOMEIP_00744] [PRS_SOMEIP_00745] [PRS_SOMEIP_00746] [PRS_SOMEIP_00747] [PRS_SOMEIP_00749] [PRS_SOMEIP_00750] [PRS_SOMEIP_00751] [PRS_SOMEIP_00752] [PRS_SOMEIP_00753] [PRS_SOMEIP_00754]

3 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the SOME/IP specification that are not included in the [1, 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 a new value on change of the value of the field.
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.

Abbreviation / Acronym:	Description:
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.

Table 3.1: Acronyms and Abbreviations

4 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 on change from the provider to the subscribers
- 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.

4.1 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.

4.1.1 Limitation

Reordering of out-of-order segments of a SOME/IP message is not supported.

4.1.2 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]

⌋(*RS_SOMEIP_00027*)

[PRS_SOMEIP_00030] is shown in the Figure 4.1.

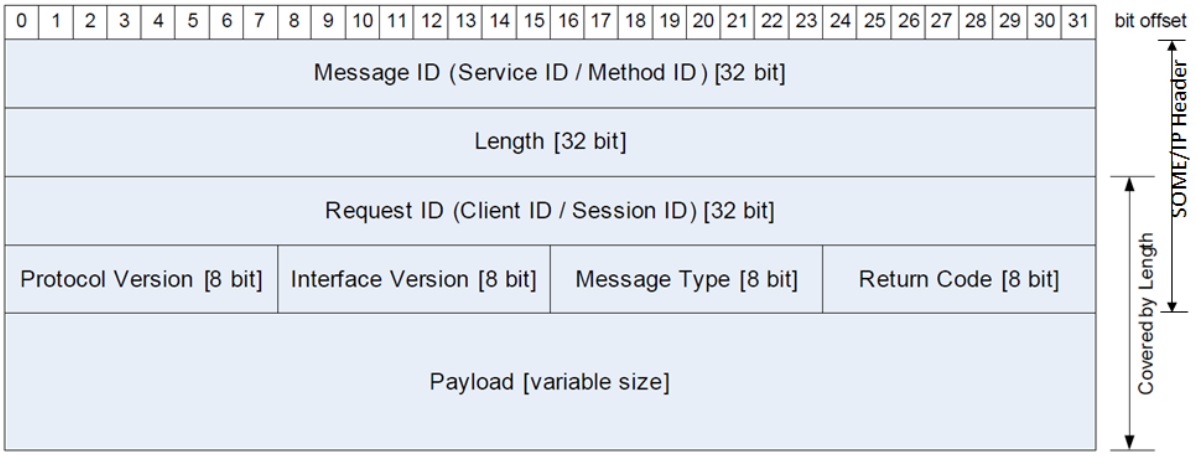

Figure 4.1: SOME/IP Header Format

[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.⌋(*RS_SOMEIP_00027*)

[PRS_SOMEIP_00941] ss shown in the Figure 4.2.

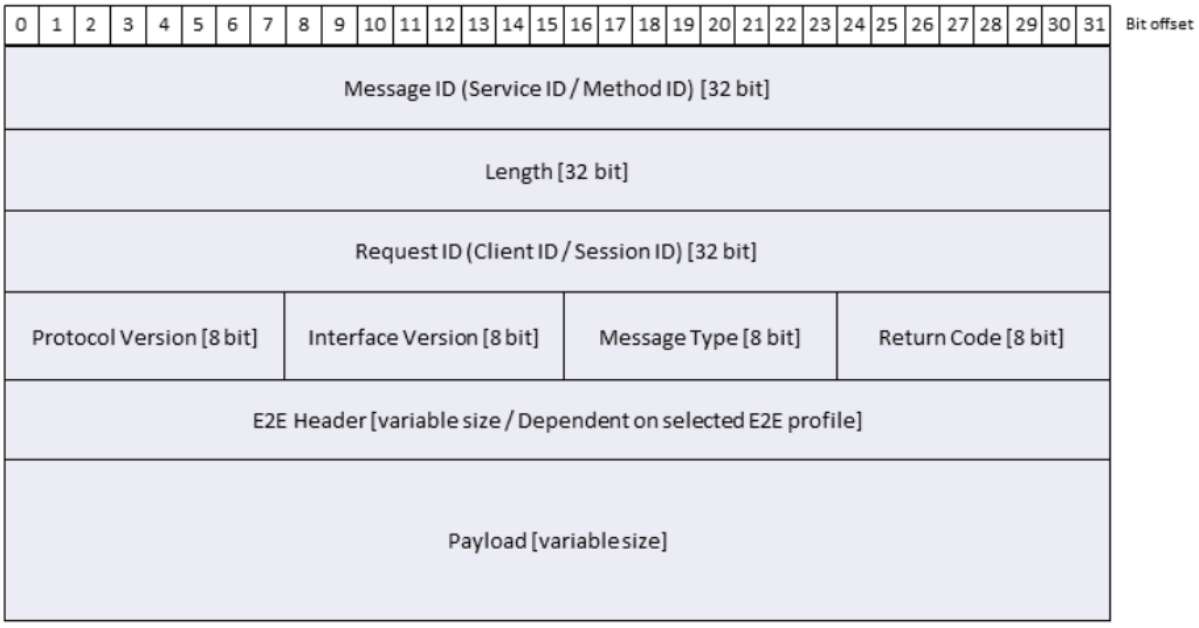

Figure 4.2: SOME/IP Header and E2E header Format

[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.⌋*(RS_SOMEIP_00027)*

4.1.2.1 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.

⌋*(RS_SOMEIP_00021, RS_SOMEIP_00022, RS_SOMEIP_00023, RS_SOMEIP_00027)*

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).

4.1.2.2 Method ID [15 Bit]

[PRS_SOMEIP_00038] ⌈Message IDs of method calls shall be structured in the ID with 2^{16} services with 2^{15} methods as shown in Table 4.1

⌋*(RS_SOMEIP_00027)*

Service ID [16 Bit]	0 [1 Bit]	Method ID [last 15 Bit]

Table 4.1: Structure of ID

4.1.2.3 Event ID [15 Bit]

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

[PRS_SOMEIP_00040] ⌈Events and notifications are transported using RPC, Events shall be structured as shown in Table 4.2

⌋*(RS_SOMEIP_00016, RS_SOMEIP_00027)*

Service ID [16 Bit]	1 [1 Bit]	Event ID [last 15 Bit]

Table 4.2: Structure of Event ID

[PRS_SOMEIP_00365] ⌈Empty eventgroups shall not be used.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00017)*

[PRS_SOMEIP_00366] ⌈Events as well as field notifiers are mapped to at least one eventgroup.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00016, RS_SOMEIP_00017, RS_SOMEIP_00018)*

4.1.2.4 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.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00040)*

4.1.2.5 Request ID [32 Bit]

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

[PRS_SOMEIP_00043] ⌈The Request ID shall be unique for a provider- and subscriber-combination (i.e. one subscription) only.⌋*(RS_SOMEIP_00002, RS_SOMEIP_00025, RS_SOMEIP_00027)*

[PRS_SOMEIP_00704] ⌈When generating a response message, the provider shall copy the Request ID from the request to the response message.⌋*(RS_SOMEIP_00027)* **Note:**
This allows the subscriber 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).⌋*(RS_SOMEIP_00025)*

Structure of the Request ID

[PRS_SOMEIP_00046] ⌈In AUTOSAR the Request ID shall be constructed of the Client ID and Session ID as shown in Table 4.3

⌋*(RS_SOMEIP_00027)*

Client ID [16 Bits]	Session ID [16 Bits]

Table 4.3: Structure of Request ID

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] ⌈The Client ID is the unique identifier for the calling client inside the ECU. The Client ID allows an ECU to differentiate calls from multiple clients to the same method.⌋*(RS_SOMEIP_00025, RS_SOMEIP_00027)*

[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.⌋*(RS_SOMEIP_00002, RS_SOMEIP_00025, RS_SOMEIP_00027)*

[PRS_SOMEIP_00532] ⌈The Client ID shall also support being unique in the overall vehicle by having a configurable prefix or fixed value (e.g. the most significant byte of Client ID being the diagnostics address or a configured Client ID for a given application/SW-C).⌋*(RS_SOMEIP_00025, RS_SOMEIP_00027)*

For example:

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

Table 4.4: Example of Client ID

[PRS_SOMEIP_00932] ⌈In case Session Handling is not active, the Session ID shall be set to `0x00`.⌋*(RS_SOMEIP_00027)*

[PRS_SOMEIP_00933] ⌈In case Session Handling is active, the Session ID shall be set to a value within the range `[0x1, 0xFFFF]`.⌋*(RS_SOMEIP_00027)*

[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]).⌋*(RS_SOMEIP_00027)*

[PRS_SOMEIP_00533] ⌈Request/Response methods shall use session handling with Session IDs. Session ID should be incremented after each call.⌋*(RS_SOMEIP_00012, RS_SOMEIP_00027)*

[PRS_SOMEIP_00521] ⌈When the Session ID reaches 0xFFFF, it shall wrap around and start again with `0x01`⌋*(RS_SOMEIP_00012, RS_SOMEIP_00027)*

[PRS_SOMEIP_00739] ⌈For request/response methods, a subscriber has to ignore a response if the Session ID of the response does not match the Session ID of the request⌋*(RS_SOMEIP_00012, RS_SOMEIP_00027)*

[PRS_SOMEIP_00935] ⌈For notification messages, a receiver shall ignore the Session ID in case Session Handling is not active.⌋*(RS_SOMEIP_00012, RS_SOMEIP_00027)*

[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.⌋*(RS_SOMEIP_00012, RS_SOMEIP_00027)*

4.1.2.6 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.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00041)*

[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.⌋ *(RS_SOMEIP_00027, RS_SOMEIP_00041)*

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

Note:
The Protocol Version itself is part of the SOME/IP Header, therefore the position of the protocol version in the header shall not be changed.

Note:
The Protocol Version shall not be increased for changes that only affect the Payload format.

[PRS_SOMEIP_00051] ⌈The Protocol Version shall be 1.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00041)*

4.1.2.7 Interface Version [8 Bit]

[PRS_SOMEIP_00053] ⌈Interface Version shall be an 8 Bit field that contains the Major Version of the Service Interface.⌋*(RS_SOMEIP_00003, RS_SOMEIP_00027)*

4.1.2.8 Message Type [8 Bit]

[PRS_SOMEIP_00055] ⌈The Message Type field is used to differentiate different types of messages and shall contain the following values as shown in Table 4.5
⌋*(RS_SOMEIP_00008, RS_SOMEIP_00027)*

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
0x80	RESPONSE	The response message
0x81	ERROR	The response containing an error

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
0xa0	TP_RESPONSE	The TP response message
0xa1	TP_ERROR	The TP response containing an error

Table 4.5: Message Types

[PRS_SOMEIP_00701] ⌈Regular request (message type 0x00) shall be answered by a response (message type 0x80), when no error occurred. If errors occur an error message (message type 0x81) shall be sent.⌋*(RS_SOMEIP_00008, RS_SOMEIP_00027)*

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.⌋*(RS_SOMEIP_00027, RS_SOMEIP_00051)* **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 4.2.1.4)

4.1.2.9 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 in Table 4.6⌋*(RS_SOMEIP_00008, RS_SOMEIP_00027)*

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).

Table 4.6: Allowed Return Codes for spcific Message Types

4.1.2.10 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. With UDP the SOME/IP payload shall be between 0 and 1400 Bytes. The limitation to 1400 Bytes is needed in order to allow for future changes to protocol stack (e.g. changing to IPv6 or adding security means). Since TCP supports segmentation of payloads, larger sizes are automatically supported.

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

4.1.3 Endianess

[PRS_SOMEIP_00368] ⌈SOME/IP Header shall be encoded in network byte order (big endian).⌋ *(RS_SOMEIP_00026)*

[PRS_SOMEIP_00369] ⌈The byte order of the parameters inside the payload shall be defined by configuration.⌋ *(RS_SOMEIP_00026)*

4.1.4 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 if the variable size data is not the last element in the serialized data stream.⌋ *(RS_SOMEIP_00028, RS_SOMEIP_00029)*

Note:
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

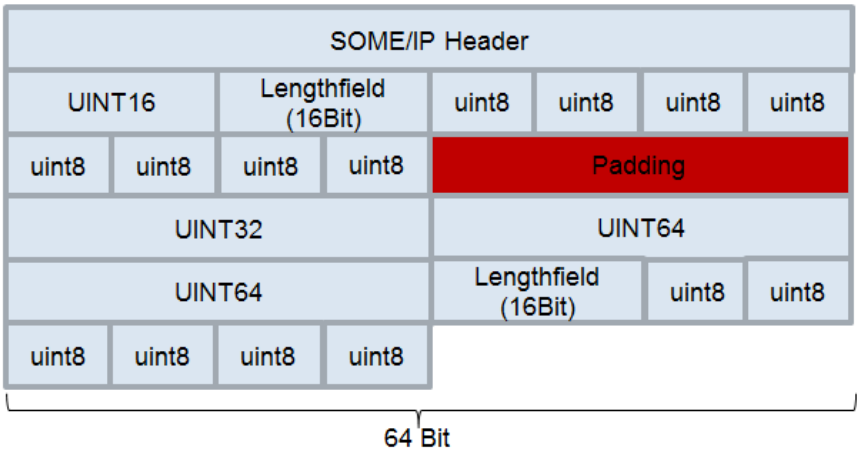

Figure 4.3: SOME/IP Padding Example 01

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

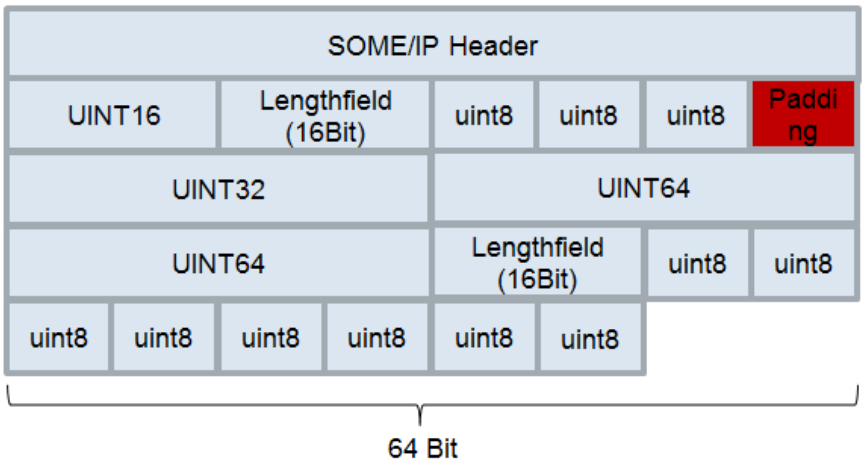

Figure 4.4: SOME/IP Padding Example 02

[PRS_SOMEIP_00569] ⌈Alignment shall always be calculated from start of SOME/IP message.⌋ *(RS_SOMEIP_00028, RS_SOMEIP_00029)*

[PRS_SOMEIP_00612] ⌈There shall be no padding behind fixed length data elements to ensure alignment of the following data.⌋ *(RS_SOMEIP_00028, RS_SOMEIP_00029)*

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 of data behind variable length data elements shall be 8, 16, 32, 64, 128 or 256. Bits.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00029)*

4.1.4.1 Basic Datatypes

[PRS_SOMEIP_00065] ⌈The following basic datatypes as shown in Table 4.7 shall be supported:

⌋*(RS_SOMEIP_00030, RS_SOMEIP_00031, RS_SOMEIP_00032)*

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)

Table 4.7: Supported basic Data Types

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.⌋*(RS_SOMEIP_00030, RS_SOMEIP_00031, RS_SOMEIP_00032)*

4.1.4.2 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:

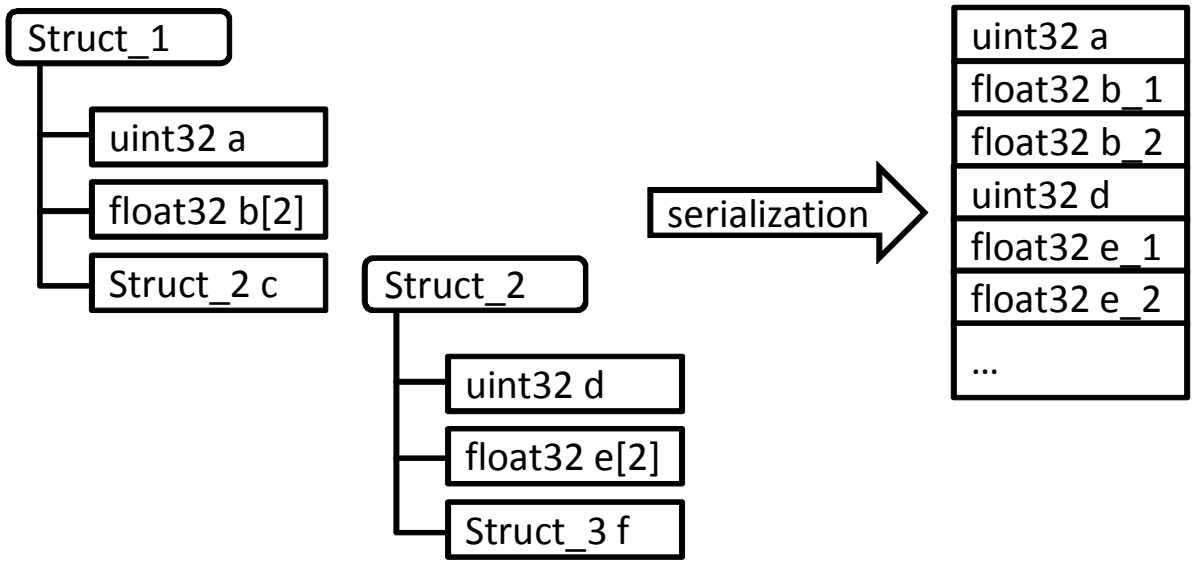

Figure 4.5: Serialization of Structs

[PRS_SOMEIP_00077] ⌈The SOME/IP implementation shall not automatically insert dummy/padding data.⌋*(RS_SOMEIP_00033)*

[PRS_SOMEIP_00079] ⌈An optional length field of 8, 16 or 32 Bit may be inserted in front of the Struct depending on the configuration.⌋*(RS_SOMEIP_00033, RS_SOMEIP_00040)*

[PRS_SOMEIP_00370] ⌈The length field of the struct shall describe the number of bytes this struct occupies for SOME/IP transport.⌋*(RS_SOMEIP_00033, RS_SOMEIP_00040)*

[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.⌋*(RS_SOMEIP_00033)*

[PRS_SOMEIP_00900] ⌈If the length is less than the sum of the lengths of all struct members 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.⌋*(RS_SOMEIP_00033)*

[PRS_SOMEIP_00712] ⌈The serialization of structs shall follow the depth-first-traversal of the structured data type.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00033)*

4.1.4.3 Structured Datatypes and Arguments with Identifier and optional members

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 member-

s/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.⌋ *(RS_SOMEIP_00050)*

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.⌋ *(RS_SOMEIP_00050)*

[PRS_SOMEIP_00231] ⌈A Data ID shall be defined either for all arguments of a method or for none of them.⌋ *(RS_SOMEIP_00050)*

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.⌋ *(RS_SOMEIP_00050)*

[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)

⌋ *(RS_SOMEIP_00050)*

Refer to the Figure 4.6 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.

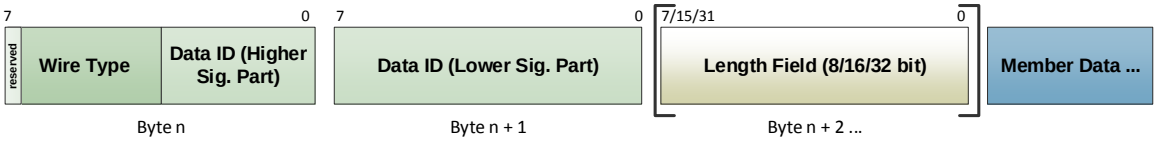

Figure 4.6: Tag Layout

[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.⌋*(RS_SOMEIP_00050)*

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] ⌈The wire type shall determine the type of the following data of the member. The value shall be assigned as shown in Table 4.8.⌋*(RS_SOMEIP_00050)*

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)

Table 4.8: Message Types

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] ⌈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.⌋*(RS_SOMEIP_00050)*

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_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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

[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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

[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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

Rationale:
The length field is required 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.⌋*(RS_SOMEIP_00040, RS_SOMEIP_00050)*

[PRS_SOMEIP_00208] ⌈If the member itself is 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].⌋*(RS_SOMEIP_00040, RS_SOMEIP_00050)*

[PRS_SOMEIP_00225] ⌈If the member itself is 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_00093], [PRS_SOMEIP_00094] and [PRS_SOMEIP_00095].⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00224] ⌈If the member itself is of type fixed length string, there shall be exactly one length field corresponding to dynamic length strings.⌋*(RS_SOMEIP_00050)*

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 member itself is 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_00107], [PRS_SOMEIP_00377] with a size of 8, 16 or 32 bit.⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00226] ⌈If the member itself is of type fixed length array, there shall be exactly one length field corresponding to dynamic length arrays.⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00228] ⌈If the member itself is 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.⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00229] ⌈If the member itself is of type union, the length field shall cover the size of the type field, data and padding bytes.⌋*(RS_SOMEIP_00050)*

Note:
For the serialization without tags, the length field of unions does not cover the type 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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

[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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

[PRS_SOMEIP_00222] ⌈The alignment of variable length data according to [PRS_SOMEIP_00611] shall always be 8 bit.⌋*(RS_SOMEIP_00029, RS_SOMEIP_00050)*

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] ⌈The size of the length field for arrays, structs, unions and string shall be greater than 0 in the configuration.⌋*(RS_SOMEIP_00050)*

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.⌋*(RS_SOMEIP_00050)*

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.⌋*(RS_SOMEIP_00050)*

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.⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00216] ⌈The serializer shall not include optional members/arguments in the serialized byte stream if they are marked as not available.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

[PRS_SOMEIP_00223] ⌈The deserializer shall ignore optional members/arguments which are not available in the serialized byte stream.⌋*(RS_SOMEIP_00050)*

[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.⌋*(RS_SOMEIP_00050)*

[PRS_SOMEIP_00218] ⌈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.⌋*(RS_SOMEIP_00050)*

[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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00050)*

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 */
```

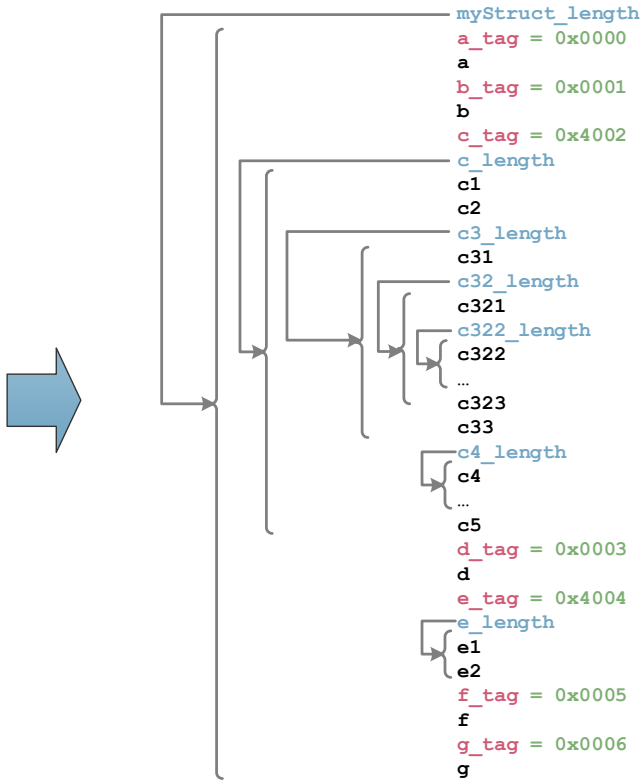

Figure 4.7: Example 01 for serializing structures with tags

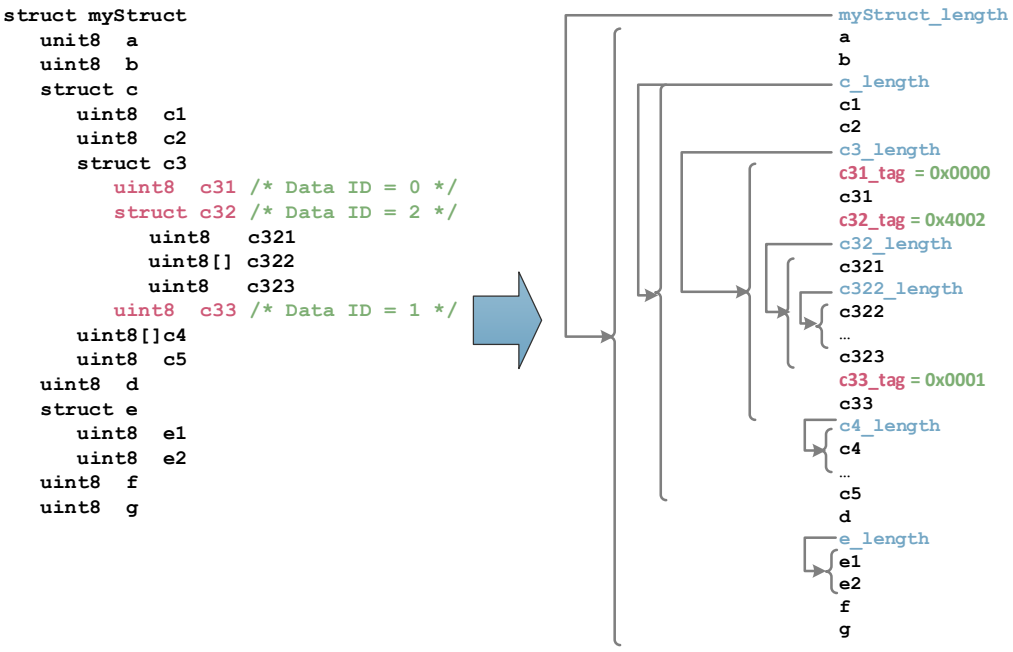

Figure 4.8: Example 02 for serializing structures with tags

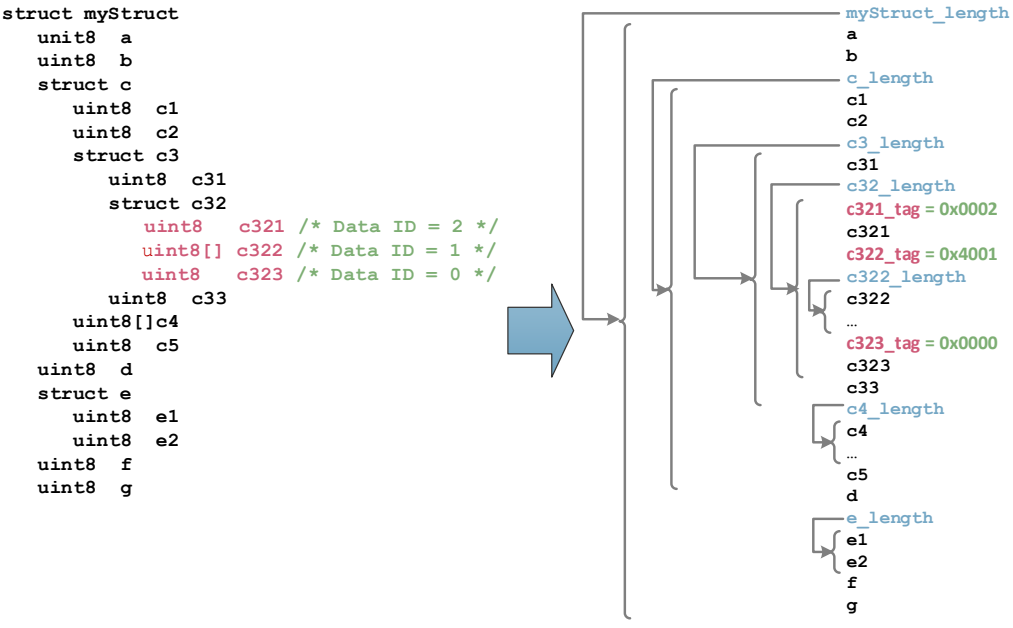

Figure 4.9: Example 03 for serializing structures with tags

Note:
In the examples Figure 4.7, Figure 4.8 and Figure 4.9 the top-level struct has a length field because it is assumed that the size of the length field is statically configured.

According to [PRS_SOMEIP_00243], the size of length field is configured for the top-level struct and is passed on to the sub elements. Thus, the top-level struct itself has a length field due to [PRS_SOMEIP_00925] and [PRS_SOMEIP_00926].

When the size of the length field is not statically configured (i.e. using wire types 5-7), the top-level struct would not have a length field.

Example for serialization of arguments with tags

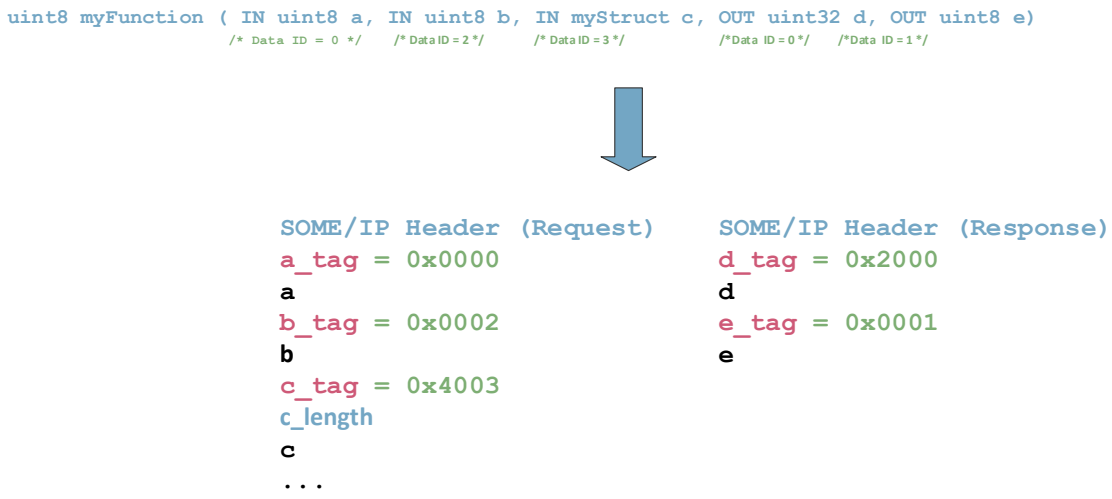

Figure 4.10: Example for serialization of arguments with tags

Note:
In the example Figure 4.10 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.

4.1.4.4 Strings

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

[PRS_SOMEIP_00372] ⌈Different Unicode encoding shall be supported including UTF-8, UTF-16BE and UTF-16LE.⌋ *(RS_SOMEIP_00038)*

[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.⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00085] ⌈UTF-16LE and UTF-16BE strings shall have an even length.⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00086] ⌈UTF-16LE and UTF-16BE strings having an odd length the last byte shall be ignored. The two bytes before shall be 0x00 bytes (termination).⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00087] ⌈All strings shall always start with a Byte Order Mark (BOM). The BOM shall be included in fixed-length-strings as well as dynamic-length strings. BOM allows the possibility to detect the used encoding.⌋ *(RS_SOMEIP_00038)*

4.1.4.4.1 Strings (fixed length)

[PRS_SOMEIP_00373] ⌈Strings shall be terminated with a "\0"-character despite having a fixed length.⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00374] ⌈The length of the string (this includes the "\0") in Bytes has to be specified in the data type definition. Fill unused space using "\0". BOM is included in the length.⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00911] ⌈If the length of a string with fixed 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.⌋ *(RS_SOMEIP_00038)*

[PRS_SOMEIP_00912] ⌈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 correctly terminated using "\0", it shall be accepted.⌋ *(RS_SOMEIP_00038)*

[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.⌋ *(RS_SOMEIP_00038)*

4.1.4.4.2 Strings (dynamic length)

[PRS_SOMEIP_00089] ⌈Strings with dynamic length shall start with a length field. The length is measured in Bytes.⌋ *(RS_SOMEIP_00039)*

[PRS_SOMEIP_00090] ⌈The length field is placed before the BOM, and the BOM is included in the length.⌋ *(RS_SOMEIP_00039)*

[PRS_SOMEIP_00091] ⌈String are terminated with a "\0".⌋ *(RS_SOMEIP_00039)*

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.⌋ *(RS_SOMEIP_00039)*

[PRS_SOMEIP_00093] ⌈Dynamic length strings shall have a length field of 8, 16 or 32 Bits. This shall be determined by configuration.⌋*(RS_SOMEIP_00039)*

[PRS_SOMEIP_00094] ⌈If not configured the length of the length field that is added in front of the string is 32 Bits (default length of length field).⌋*(RS_SOMEIP_00039, RS_SOMEIP_00040)*

[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.⌋*(RS_SOMEIP_00039)*

[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.⌋*(RS_SOMEIP_00039)*

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 4.1.4.5.2). Please note that this requires the full string handling (e.g. endianness conversion) to be done in the applications.

This can also be achieved by setting the attribute `implementsLegacyStringSerialization` to true. In CP this attribute is configured in `SOMEIPTransformationISignalProps`, in AP it is configured in `ApSomeipTransformationProps`.

NOTE! This attribute is not future safe and will be removed in an upcoming AUTOSAR release!

Therefore to be forward compatible, plain dynamic length arrays should preferably be used in this case.

4.1.4.5 Arrays

4.1.4.5.1 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] ⌈Arrays with fixed length may start with an optional length field.⌋*(RS_SOMEIP_00036)*

[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.⌋*(RS_SOMEIP_00036)*

[PRS_SOMEIP_00918] ⌈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.⌋ *(RS_SOMEIP_00036)*

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 preceed the first element (see [PRS_SOMEIP_00944].⌋ *(RS_SOMEIP_00035, RS_SOMEIP_00036)*

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 4.11.

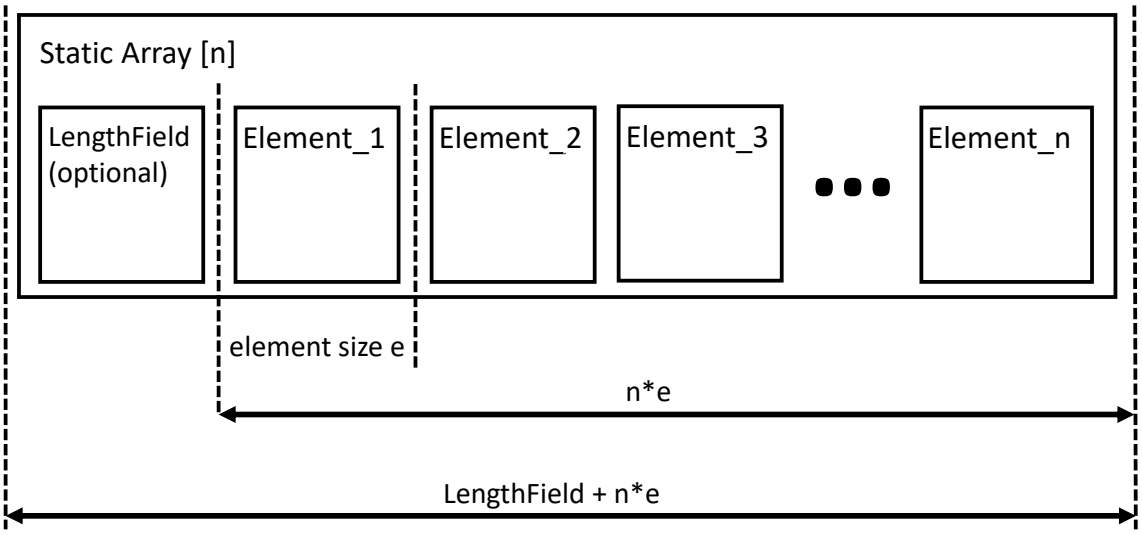

Figure 4.11: One-dimensional array (fixed length)

Multidimensional

[PRS_SOMEIP_00101] ⌈The serialization of multidimensional arrays follows the in-memory layout of multidimensional arrays in the programming language (row-major order).⌋ *(RS_SOMEIP_00028, RS_SOMEIP_00035, RS_SOMEIP_00036)*

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 4.12.

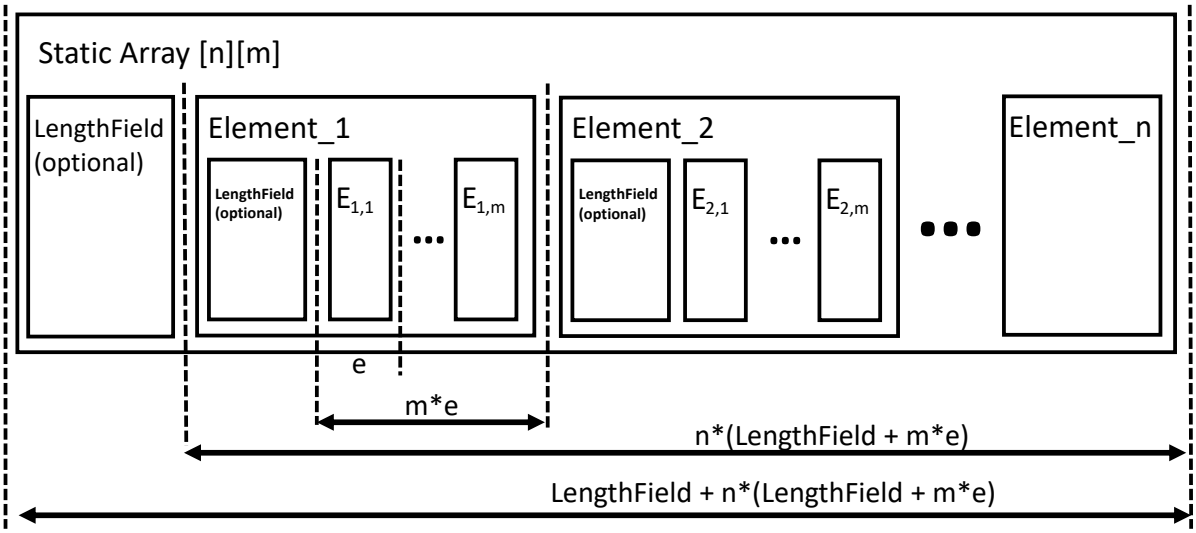

Figure 4.12: Multidimensional array (fixed length)

4.1.4.5.2 Dynamic Length Arrays

[PRS_SOMEIP_00375] ⌈The layout of arrays with dynamic length shall be based on the layout of fixed length arrays.⌋*(RS_SOMEIP_00037)*

[PRS_SOMEIP_00376] ⌈An optional length field at the beginning of an array should be used to specify the length of the array in Bytes.⌋*(RS_SOMEIP_00037)*

[PRS_SOMEIP_00107] ⌈The length field shall have a length of 0, 8, 16 or 32 Bits. This shall be determined by configuration.⌋*(RS_SOMEIP_00037)*

[PRS_SOMEIP_00377] ⌈The length does not include the size of the length field.⌋*(RS_SOMEIP_00037)*

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 4.13 and Figure 4.14.

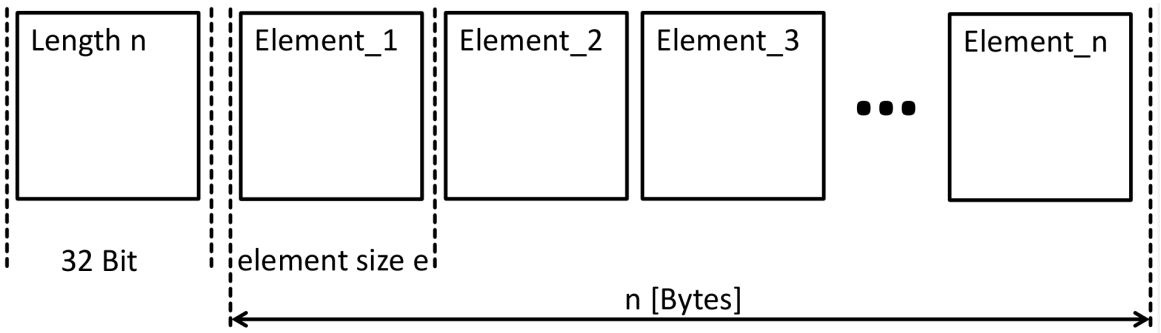

Figure 4.13: One-dimensional array (dynamic length)

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 4.14 shows the structure of a Multidimensional Array of dynamic length.

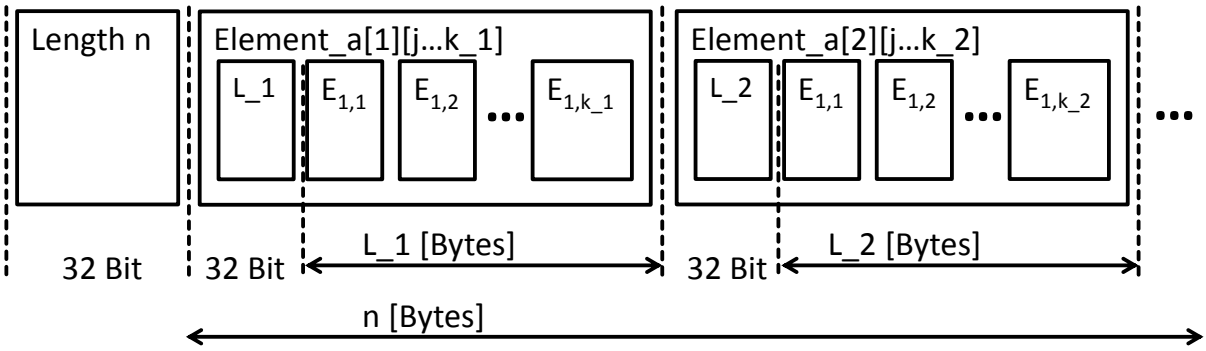

Figure 4.14: Multidimensional array (dynamic length)

[PRS_SOMEIP_00114] ⌈In multidimensional arrays every sub array of different dimensions shall have its own length field.⌋ *(RS_SOMEIP_00037)*

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 4.14. 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_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.⌋*(RS_SOMEIP_00037)*

[PRS_SOMEIP_00945] ⌈If not configured the length of the length field that is added in front of the dynamic length array is 32 Bits (default length of length field).⌋*(RS_SOMEIP_00037, RS_SOMEIP_00040)*

4.1.4.6 Enumeration

[PRS_SOMEIP_00705] ⌈Enumerations are not considered in SOME/IP. Enumerations shall be transmitted as unsigned integer datatypes.⌋*(RS_SOMEIP_00030, RS_SOMEIP_00033)*

4.1.4.7 Bitfield

[PRS_SOMEIP_00300] ⌈Bitfields shall be transported as unsigned datatypes uint8/uint16/uint32.⌋*(RS_SOMEIP_00033, RS_SOMEIP_00030)*

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

4.1.4.8 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.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00119] ⌈A union shall consist of a length field, type selector and the payload as shown in Table 4.9:⌋*(RS_SOMEIP_00034)*

Length field [32 bit]
Type field [32 bit]
Data including padding [sizeof(padding) = length - sizeof(data)]

Table 4.9: Default Serialization of Unions

[PRS_SOMEIP_00126] ⌈The length field shall define the size of the data and padding in bytes and does not include the size of the length field and type field.⌋*(RS_SOMEIP_00034)*

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 and shall be 32, 16, 8, or 0 bits⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00122] ⌈A length of the length field of 0 Bit means that no length field will be written to the PDU.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00123] ⌈If the length of the length field is 0 Bit, all types in the union shall be of the same length.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00129] ⌈The type field shall specify the data type of the data.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00127] ⌈The length of the type field shall be defined by configuration and shall be 32, 16, or 8 bits.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00906] ⌈Possible values of the type field shall be defined by the configuration for each union separately.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00907] ⌈The value 0 of the type field shall be reserved for the NULL type.⌋*(RS_SOMEIP_00024, RS_SOMEIP_00034)*

Note:
This denotes an empty union.

[PRS_SOMEIP_00908] ⌈Whether NULL is allowed for a union or not shall be defined by configuration.⌋*(RS_SOMEIP_00034)*

[PRS_SOMEIP_00130] ⌈The payload is serialized depending on the type in the type field.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00034)*

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 4.10.

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

Table 4.10: Example: uint8

A uint16 will be serialized like shown in Table 4.11.

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

Table 4.11: Example: uint16

[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.⌋ *(RS_SOMEIP_00034)*

[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.⌋ *(RS_SOMEIP_00034)*

4.2 Specification of SOME/IP Protocol

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

4.2.1 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 6 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.⌋ *(RS_SOMEIP_00015)*

For details of see Chapter 4.2.1.3

[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).⌋ *(RS_SOMEIP_00010)*

[PRS_SOMEIP_00142] ⌈The receiving SOME/IP implementation shall be capable of receiving unaligned SOME/IP messages transported by UDP or TCP.⌋*(RS_SOMEIP_00010)*

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.⌋*(RS_SOMEIP_00010)*

[PRS_SOMEIP_00141] ⌈Each SOME/IP payload shall have its own SOME/IP header.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00027)*

[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

⌋*(RS_SOMEIP_00010, RS_SOMEIP_00027)*

4.2.1.1 UDP Binding

[PRS_SOMEIP_00139] ⌈The UDP binding of SOME/IP shall be achieved by transporting SOME/IP messages in UDP packets.⌋*(RS_SOMEIP_00010)*

[PRS_SOMEIP_00137] ⌈SOME/IP protocol shall not restrict the usage of UPD fragmentation.⌋*(RS_SOMEIP_00010)*

[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.⌋*(RS_SOMEIP_00010)*

[PRS_SOMEIP_00942] ⌈The client and server shall use a single UDP multicast connection for all events, and notifications of a Service-Instance which are configured to be communicated using UDP multicast.⌋*(RS_SOMEIP_00010)*

4.2.1.2 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, outstanding requests shall be handled as timeouts.⌋*(RS_SOMEIP_00010)* 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.⌋*(RS_SOMEIP_00010)*

[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.⌋*(RS_SOMEIP_00010)*

The client is responsible for reestablishing 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.⌋*(RS_SOMEIP_00010)*

[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).⌋*(RS_SOMEIP_00010)*

[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.⌋*(RS_SOMEIP_00010)*

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.⌋*(RS_SOMEIP_00010)*

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

- for communincation 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 communincation 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`

⌋(*RS_SOMEIP_00010*)

The layout of the Magic Cookie Messages is shown in Figure 4.15.

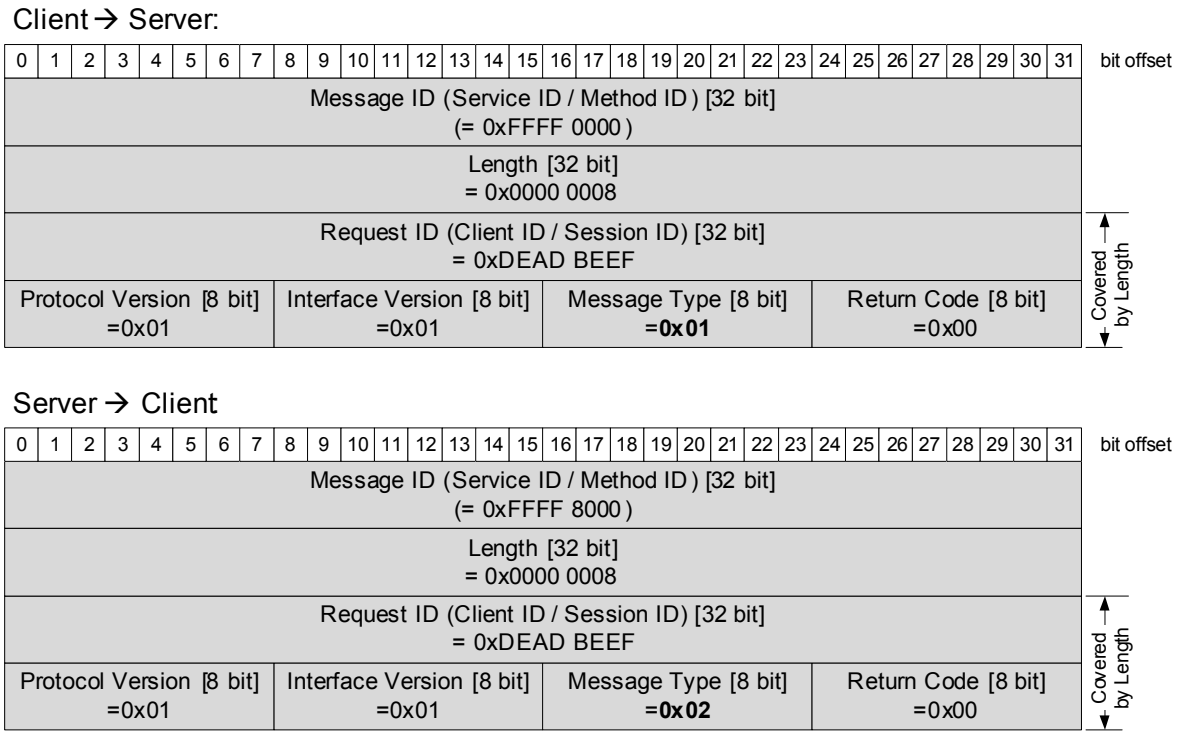

Figure 4.15: SOME/IP Magic Cookie Message for SOME/IP

4.2.1.3 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.⌋ *(RS_SOMEIP_00015)*

[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, multiple Service-Instances of the same Service on one single ECU shall use different ports per Service-Instance.⌋ *(RS_SOMEIP_00015)*

Rationale: While Instance IDs are used for Service Discovery, they are not contained in the SOME/IP header.

A Service Instance can be identified through the combination of the Service ID combined with the socket (i.e. IP-address, transport protocol (UDP/TCP), and port number). It is recommended that instances use the same port number for UDP and TCP. If a service instance uses UDP port x, only this instance of the service and not another instance of the same service should use exactly TCP port x for its services.

4.2.1.4 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).⌋*(RS_SOMEIP_00010, RS_SOMEIP_00012)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00012)*

[PRS_SOMEIP_00722] ⌈SOME/IP-TP segments shall have the TP-Flag of the Message Type set to 1.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00011)*

[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]

⌋*(RS_SOMEIP_00010, RS_SOMEIP_00027)*

SOME-IP-TP-Header is as shown in Figure 4.16.

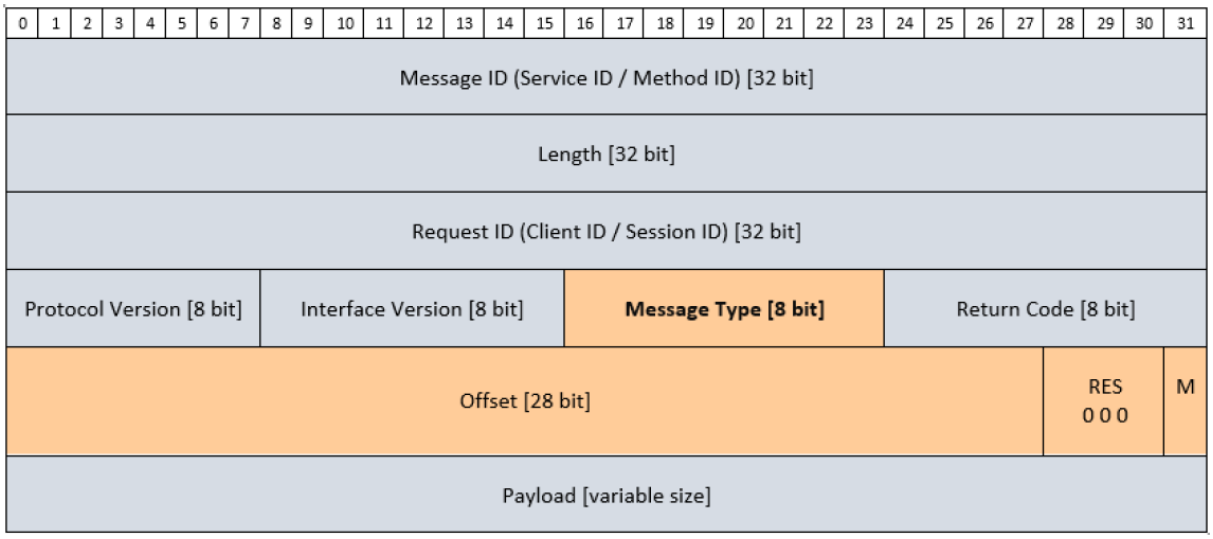

Figure 4.16: SOME/IP TP header

[PRS_SOMEIP_00931] ⌈SOME/IP-TP Header shall be encoded in network byte order (big endian).⌋ *(RS_SOMEIP_00027)*

[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.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00027)* **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.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00027)*

[PRS_SOMEIP_00726] ⌈The Reserved Flags shall be set to 0 by the sender and shall be ignored (and not checked) by the receiver.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00027)*

[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.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00027)*

[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.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00027)* **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.⌋ *(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00730] ⌈Since UDP-based SOME/IP messages are limited to 1400 bytes payload, the maximum length of a segment that is correctly aligned is 1392 bytes.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00029, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)* **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.

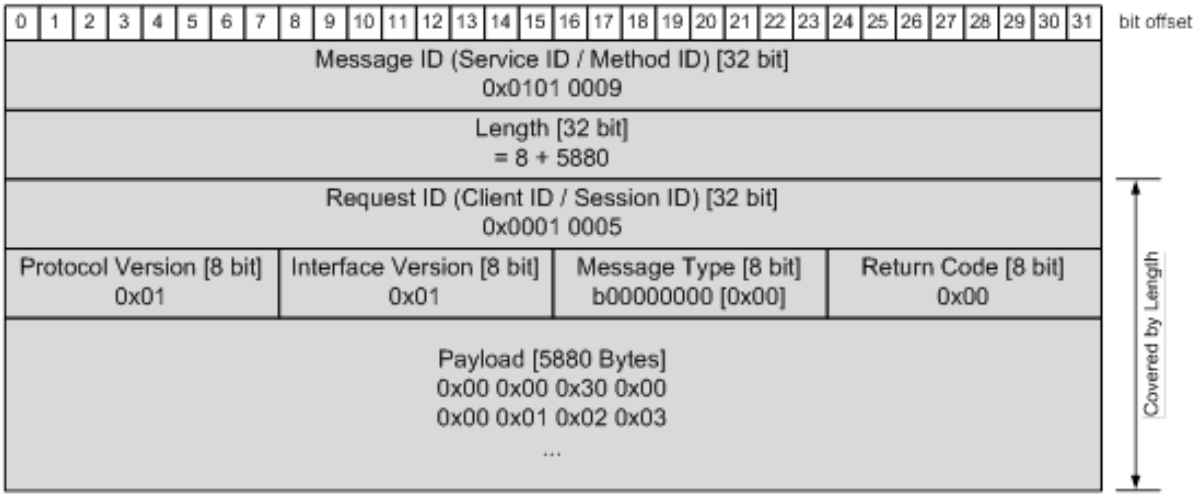

Figure 4.17: Example: Header of Original SOME/IP message

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

Figure 4.18: 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':

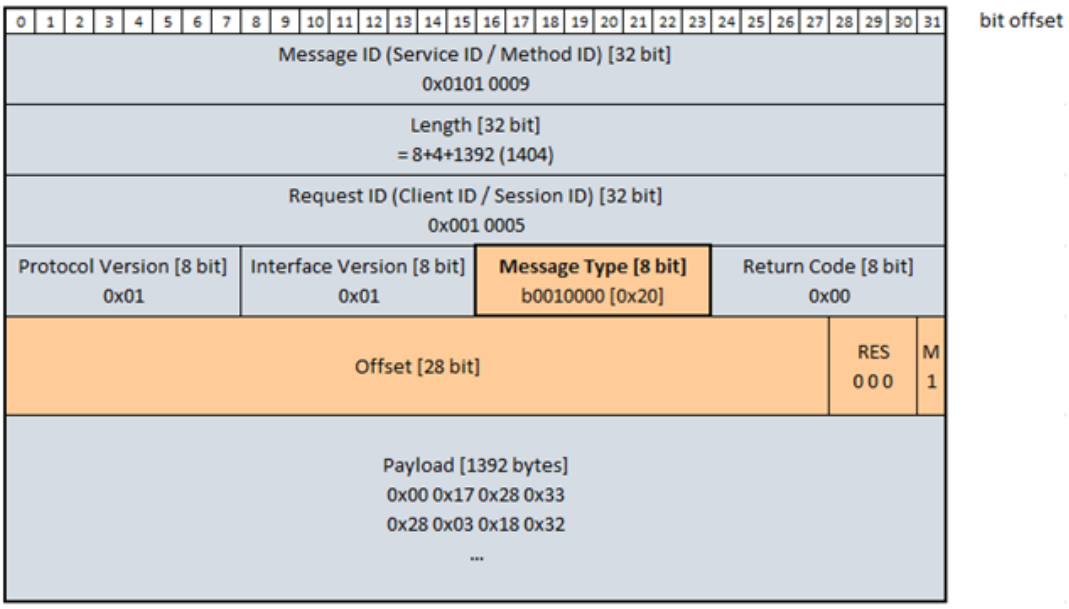

Figure 4.19: Example: Header of the SOME/IP segments

- 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'.

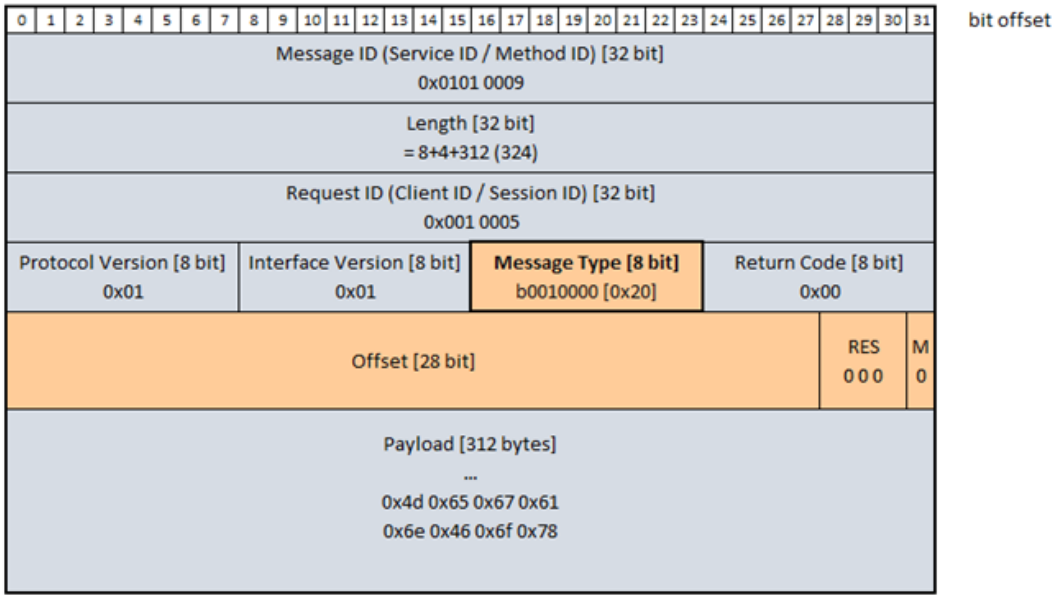

Figure 4.20: Example: Header of the last SOME/IP segment

Sender specific behavior

[PRS_SOMEIP_00732] ⌈The sender shall segment only messages that were configured to be segmented.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00733] ⌈The sender shall send segments in ascending order.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00735] ⌈The sender shall try to maximize the size of segments within limitations imposed by this specification.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00736] ⌈The sender shall not send overlapping or duplicated segments.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

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).⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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".⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00741] ⌈The Session ID shall be used to detect the next original message to be reassembled.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00743] ⌈The receiver should only reassemble up to its configured buffer size and skip the rest of the message.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00744] ⌈Only correctly reassembled message of up to the configured size shall be passed to an application.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*
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_00745] ⌈The Return Code of the last segment used for reassembly shall be used for the reassembled message.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00747] ⌈The receiver shall support reassembly of segments that are received in ascending and descending order.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00749] ⌈When a missing segment is detected during assembly of a SOME/IP message, the current assembly process shall be canceled.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)* **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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[PRS_SOMEIP_00752] ⌈The receiver shall correctly reassemble overlapping and duplicated segments by overwriting based on the last received segment.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)*

[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.⌋*(RS_SOMEIP_00010, RS_SOMEIP_00051)* **Note:**
This means that buffer overflows or other malfunction shall be prevented by the receiving code.

4.2.2 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`

⌋*(RS_SOMEIP_00007)*

[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.⌋*(RS_SOMEIP_00028, RS_SOMEIP_00007)*

[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 the return code of called method or in case of an Error message to a valid error code.

⌋(*RS_SOMEIP_00007*)

For the valid retrun codes, see Table 4.12.

[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.⌋(*RS_SOMEIP_00028, RS_SOMEIP_00007*)

[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.⌋(*RS_SOMEIP_00007*)

[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.⌋(*RS_SOMEIP_00007*)

4.2.3 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`

⌋(*RS_SOMEIP_00006*)

[PRS_SOMEIP_00171] ⌈Fire & Forget messages shall not return an error. Error handling and return codes shall be implemented by the application when needed.⌋*(RS_SOMEIP_00006)*

4.2.4 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`

⌋*(RS_SOMEIP_00004)*

[PRS_SOMEIP_00926] ⌈The payload of the notification message shall consist of the serialized data of the event.⌋*(RS_SOMEIP_00004)*

[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.⌋*(RS_SOMEIP_00042)*

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

4.2.4.1 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.

4.2.5 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.⌋ *(RS_SOMEIP_00009)*

[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.⌋ *(RS_SOMEIP_00009)*

[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.⌋ *(RS_SOMEIP_00009)*

[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.⌋ *(RS_SOMEIP_00009)*

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.⌋ *(RS_SOMEIP_00002, RS_SOMEIP_00009)*

[PRS_SOMEIP_00183] ⌈The notifier shall send an event message that transports the value of a field on change and follows the rules for events.⌋ *(RS_SOMEIP_00005, RS_SOMEIP_00009)*

4.2.6 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 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.⌋*(RS_SOMEIP_00008)*

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.⌋*(RS_SOMEIP_00008)*

[PRS_SOMEIP_00903] ⌈If more detailed error information need to be transmitted, the payload of the Error Message (Message Type `0x81`) shall be filled with error specific data, e.g. an exception string. Error Messages shall be sent instead of Response Messages.⌋*(RS_SOMEIP_00008)*

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 4.1.2)

[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.⌋*(RS_SOMEIP_00008)*

[PRS_SOMEIP_00905] ⌈All other messages than `0x80` and `0x81` shall set this field to `0x00`.⌋*(RS_SOMEIP_00008)*

For message type see Chapter 4.1.2.8.

4.2.6.1 Return Code

[PRS_SOMEIP_00187] ⌈The return code shall be UINT8.⌋*(RS_SOMEIP_00008)*

[PRS_SOMEIP_00191] ⌈Currently defined Return Codes are shown in Table 4.12⌋ *(RS_SOMEIP_00008, RS_SOMEIP_00024)*

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	Service ID and Method ID are known. Application not running.
0x05	E_NOT_REACHABLE	System running the service is not reachable (internal error code only).
0x06	E_TIMEOUT	A timeout occurred (internal error code only).
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. REQUEST_NO_RETURN for a method defined as REQUEST).
0x0b	E_E2E_REPEATED	Repeated E2E calculation error
0x0c	E_E2E_WRONG_SEQUENCE	Wrong E2E sequence error
0x0d	E_E2E	Not further specified E2E error
0x0e	E_E2E_NOT_AVAILABLE	E2E not available
0x0f	E_E2E_NO_NEW_DATA	No new data for E2E calculation present.
0x10 - 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.

Table 4.12: Return Codes

Generation and handling of return codes shall be configurable.

[PRS_SOMEIP_00539] ⌈A SOME/IP error message (i.e. return code `0x01 - 0x1f`) shall not be answered with an error message.⌋ *(RS_SOMEIP_00008)*

4.2.6.2 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.⌋ *(RS_SOMEIP_00008)*

[PRS_SOMEIP_00189] ⌈The receiver of a SOME/IP message shall not return an error message for fire&forget methods.⌋ *(RS_SOMEIP_00008)*

[PRS_SOMEIP_00537] ⌈The receiver of a SOME/IP message shall not return an error message for events/notifications and fire&forget methods if the Message Type is set incorrectly to Request or Response.⌋ *(RS_SOMEIP_00008)*

[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.⌋ *(RS_SOMEIP_00008)*

4.2.6.3 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].⌋ *(RS_SOMEIP_00008, RS_SOMEIP_00014)*

[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.⌋ *(RS_SOMEIP_00008, RS_SOMEIP_00014)*

[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.⌋ *(RS_SOMEIP_00008, RS_SOMEIP_00014)*

An overview of the error processing is shown in Figure 4.21.

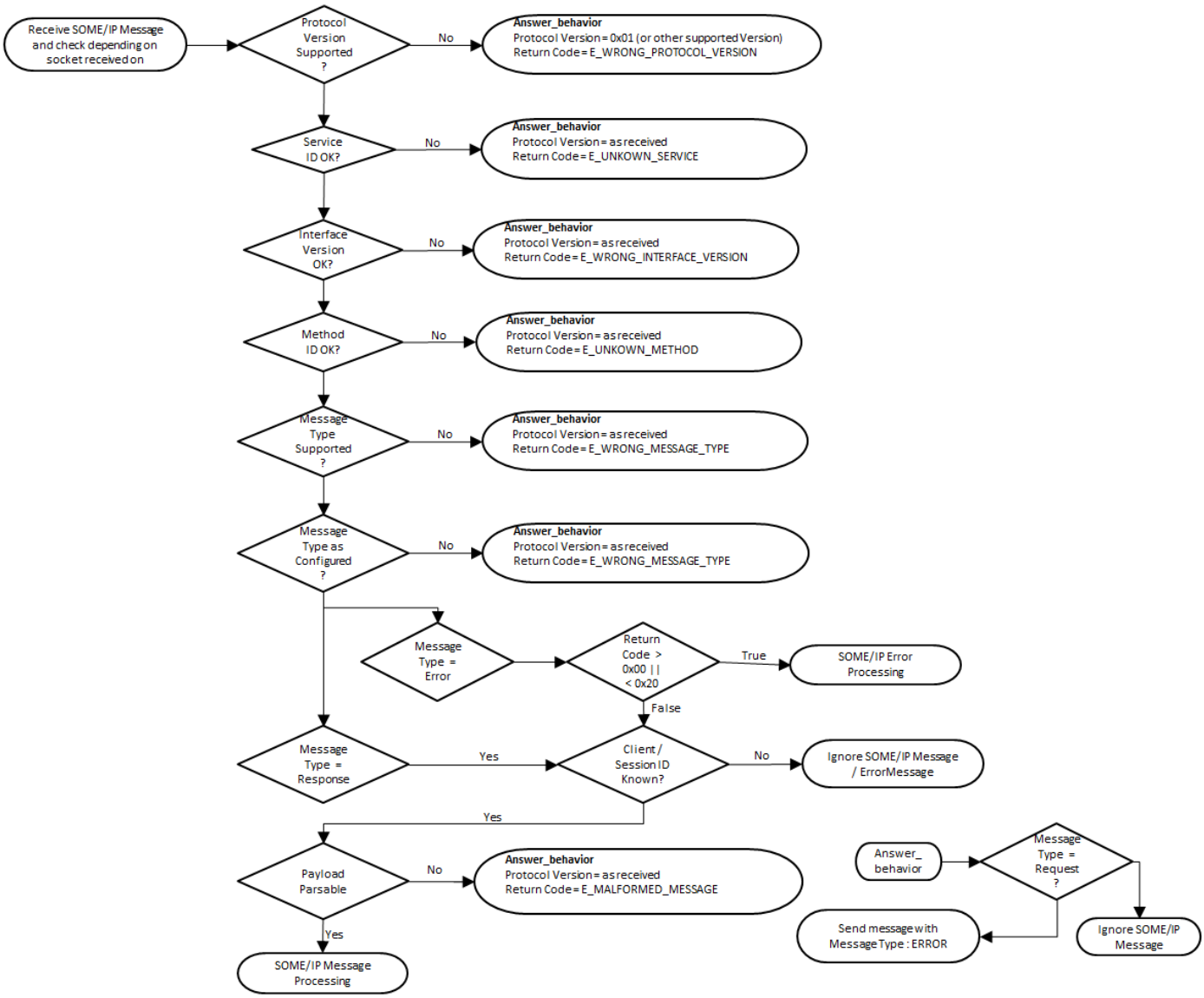

Figure 4.21: Message Validation and Error Handling in SOME/IP

[PRS_SOMEIP_00614] ⌈When one of the errors specified in [PRS_SOMEIP_00195] occurs while receiving SOME/IP messages over TCP, the receiver shall check the TCP connection and shall restart the TCP connection if needed.⌋*(RS_SOMEIP_00008, RS_SOMEIP_00014)* Rational:
Checking the TCP connection might include the following:

- Checking whether data is received for e.g. other Eventgroups.
- Sending out a Magic Cookie message and waiting for the TCP ACK.
- Reestablishing the TCP connection

4.2.6.4 Communication Errors and 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 4.22 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.

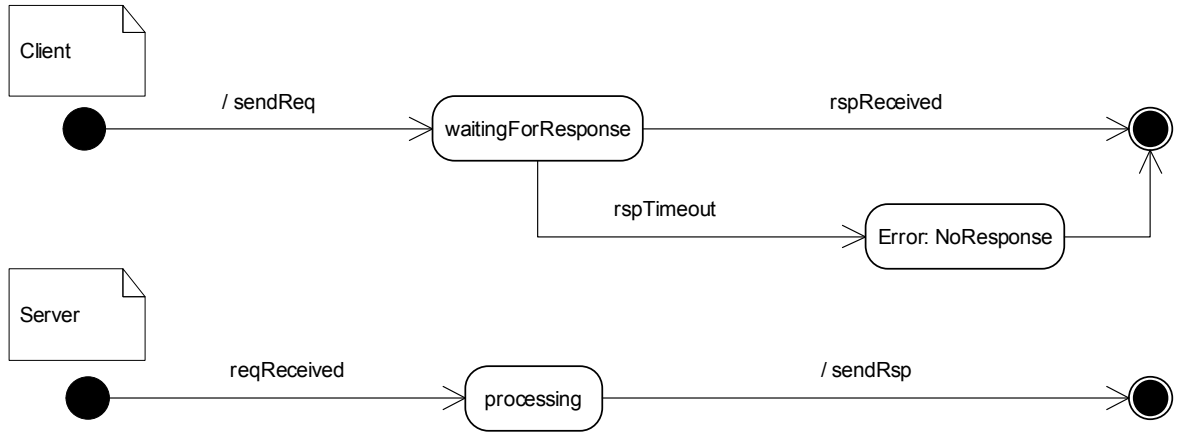

Figure 4.22: State Machines for Reliability "Maybe"

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

4.3 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 behaviour
- required by application design

⌋*(RS_SOMEIP_00003)*

Note: The Interface Version shall not be increased for compatible changes in the Payload format.

[PRS_SOMEIP_00938] ⌈The rules in Table 4.13 shall define the compatibility of changes of the payload format. For complex data types the rules shall be applied recursively. X denotes a compatible change, an empty cell denotes an incompatible change.⌋*(RS_SOMEIP_00003)*

Note:
This table is 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.

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	
	Provide	Require	Provide	Require	Provide	Require	Provide	Require
Add a struct member **not** to the end of the struct					X		X	X
Add a struct member to the end of the toplevel struct	X		X		X		X	X
Add a struct member to the end of a sub-struct			X		X		X	X
Remove struct member **not** from the end of the struct						X	X	X
Remove struct member from the end of the toplevel struct		X		X		X	X	X
Remove struct member from the end of a sub-struct				X		X	X	X
Reorder struct members					X	X	X	X
Change the non-highest union member (redefine or remove)								
Add a new union member with previously unused type selector		X		X		X		X
Remove union member with highest type selector	X		X		X		X	
Change of data type: • to a larger one (e.g. uint8 to uint16) • to a smaller one (e.g. uint16 to uint8 • to a semantically different one (e.g. integer to struct, integer to float, string to string with different character size) • byte order • number of dimensions of arrays • size of length field of array, struct or union type selector								
Add new enumeration values		X		X		X		X
Change existing enumeration values								
Remove enumeration values	X		X		X		X	
Change length of fixed size string or array			N/A	N/A	N/A	N/A	N/A	N/A
Decrease maximum length of variable size string	N/A	N/A	X		X		X	
Increase maximum length of variable size string	N/A	N/A		X		X		X
Change maximum length of variable size array	N/A	N/A	X	X	X	X	X	X
Add argument **not** to the end of the argument list of a method request Remove argument **not** from the end of the argument list of a method reponse						X	X	X
Add argument to the end of the argument list of a method request Remove argument from the end of the argument list of a method reponse		X		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	
	Provide	Require	Provide	Require	Provide	Require	Provide	Require
Remove argument **not** from the end of the argument list of a method request Add argument **not** from the end of the argument list of a method response					X		X	X
Remove argument from the end of the argument list of a method request Add argument from the end of the argument list of a method response	X		X		X		X	X
Reorder arguments of methods					X	X	X	X
Change optionality of argument	N/A	N/A	N/A	N/A	N/A	N/A		
Change the return type of a method (e.g void to uint8)								
Add return codes of a method		X		X		X		X
Remove return codes of a method	X		X		X		X	
Change of the name of a service interface, method or event	X	X	X	X	X	X	X	X
Add event ot eventgroup	X		X		X		X	
Remove event from eventgroup		X		X		X		X
Add setter or getter to a field Remove notifier from a field	X		X		X		X	
Remove setter or getter from a field Add notifier to a field		X		X		X		X
Extend service interface by new method, event or field	X		X		X		X	
Remove method, event or field from a service interface		X		X		X		X
Change Method ID or Event ID								
Change data ID of argument Reuse data ID of repviously removed argument	N/A	N/A	N/A	N/A				

Table 4.13: Payload Compatibility Rules

5 Configuration Parameters

Configuration Parameters are not handled and described in this document.

6 Protocol usage and guidelines

6.1 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.

6.2 Transporting CAN and FlexRay Frames

SOME/IP should allow to tunnel CAN and FlexRay frames. However, the Message ID space needs to be coordinated between both use cases. The full SOME/IP Header shall be used for transporting/tunneling CAN/FlexRay.

6.3 Insert Padding for structs

If padding is necessary, the designer of the interface shall Insert reserved/padding elements in the the definition of the data types if needed for alignment, since the SOME/IP implementation shall not automatically add such padding.

7 References

References

[1] Glossary
AUTOSAR_TR_Glossary