Entwicklungsziel/ *development objectives* PEP-Sync.	**Lastenheft** ***Requirements Specification*** Dokumentenart / Type of document	vertraulich/ *confidential* Hinweis zur Geheimhaltung / Note on confidentiality
BMW Group	SOME / IP Serialisierung	10162833 – 000 – 01 PDM-Dokumentnummer - Teildokument – Version PDM document number – document section – version
		G11 Erstverw.-Typ / First used in model
		08.08.2013 Datum PDM-Status / Date PDM Status
		ungültig / *Invalid* PDM-Status

Fortsetzung/*Continued* Seite/*Page* 2 bis/*to* 23

Funktion ***Function***	**Abteilung** ***Department***	**Name / Unterschrift** ***Name/ Signature***	**Datum** ***Date***
Ergebnisverantwortlich *responsible*	EI-31	Lars Voelker	29.11.2011
Qualitätsprüfung bestätigt *quality check approved*	EI-31	Lutz Rothhardt	30.11.2011
Genehmigt *approved*	EI-31	Lutz Rothhardt	30.11.2011
Gesehen	EI-31	Richard Bogenberger	30.11.2011
Lastenheft akzeptiert *Requirements specification approved*	Lieferant *Supplier*		

Änderungsdokumentation – Change documentation

Änderungs-Nr. / NAEL	Version / ZI	Abschnitt	Kurzbeschreibung	Datum	Name
	01		Initialversion E-Ziel-frei	2011-11-29	L. Völker

1 Introduction

This document specifies the Scalable serviceOriented middlewarE over IP (SOME/IP) – an automotive/embedded RPC mechanism and the underlying serialization / wire format.

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
- Is compatible with AUTOSAR at least on the wire-format level; i.e. can communicate with PDUs AUTOSARs can receive and send without modification to the AUTOSAR standard. The mappings within AUTOSAR must be chosen according to the SOME/IP specification.
- Provides the features required by automotive use-cases
- Is scalable from tiny to large platforms
- Can be implemented on different operating system (i.e. AUTOSAR, GENIVI, and OSEK) and even embedded devices without operating system

The basic feature set of the SOME/IP wire format will be supported starting with AUTOSAR 4.0.3. This allows AUTOSAR to parse the RPC PDUs and transport the signals to the application. However, this basic feature is not enough for more sophisticated use cases (e.g. Infotainment applications).

As a consequence this specification defines several feature sets. The feature set "basic" is fully compatible to AUTOSAR 4.0.3. The other feature sets are in progress to be integrated into AUTOSAR. The goal is to increase the compatibility towards higher sophisticated feature sets. It is however possible to use these features in non-AUTOSAR nodes or to implement them inside the AUTOSAR application with a carefully designed interface (see Section 4) and suitable tool chain.

For ECUs not using AUTOSAR the complete feature set can be supported as of today but a limited set of features can be used in the communication with AUTOSAR ECUs.

1.1 Definition of terms

- Method – a method, procedure, function, or subroutine that can be called/invoked
- Parameters – input, output, or input/output arguments of a method
  - Input/output arguments are arguments shared for input and output
- Remote Procedure Call (RPC) – a method call from one ECU to another that is transmitted using messages
- Server – the software component that offers a method
- Client – the software component that invokes a method
- 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
- Fire&Forget Communication – a RPC call that consists only of a request message
- Event – a Fire&Forget callback that is only invoked on changes or cyclic
- Service – a logical combination of methods and events
- Service Interface – the formal specification of the service including its methods and events
- Union or Variant – a data structure that can dynamically assume different data types

2 Specification of the SOME/IP on-wire format

Serialization describes the way data is represented in protocol data units (PDUs) transported over an IP-based automotive in-vehicle network.

2.1 Transport Protocol

SOME/IP may be transported using UDP or TCP. The port numbers for SOME/IP have to be defined locally from the private port range 49152-65535 until further notice. When used in a vehicle the OEM will specify the ports used in the interface specification.

It is recommended to use UDP for as many messages as possible and see TCP as fall-back for message requiring larger size. UDP allows the application to better control the timeouts and behavior when errors occurring.

2.1.1 Message Length Limitations

In combination with regular Ethernet, IPv4 and UDP can transport packets with up to 1472 Bytes of data without fragmentation, while IPv6 takes another 20 Bytes. Especially for small systems fragmentation shall be avoided, so the SOME/IP header and payload should be of limited length. The possible usage of security protocols further limits the maximal size of SOME/IP messages.

When using UDP as transport protocol SOME/IP messages may use up to 1416 bytes for the SOME/IP header and payload.

The usage of TCP allows for larger streams of data, which may be used for the SOME/IP header and payload. However, current transport protocols for CAN and FlexRay as well as AUTOSAR limit messages to 4095 Bytes. When compatibility to CAN or FlexRay has to be achieved, SOME/IP messages including the SOME/IP header must not exceed 4095 Bytes.

See also Section 2.3.9 for payload length.

2.1.1.1 AUTOSAR restrictions

See AUTOSAR SWS COM Chapter 7.6.

2.2 Endianess

All RPC-Headers must be encoded with in network byte order (big endian) [RFC 791]. The byte order of the parameters inside the payload must be defined by the interface definition (i.e. FIBEX) and should be in network byte order when possible and if no other byte order is specified.

2.3 Header

For interoperability reasons the header layout shall be identical for all scales of the SOME/IP and is shown in Figure 1. The fields are presented in transmission order; i.e. the fields on the top left are transmitted first. In the following sections the different header fields and their usage is being described.

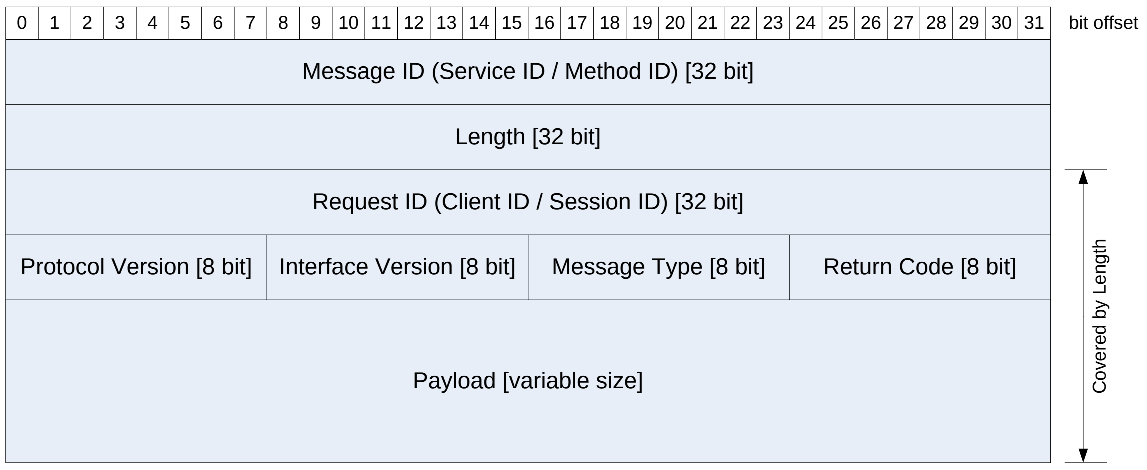

Figure 1 SOME/IP Header Format

2.3.1 IP-Address / port numbers

The Layout in Figure 1 shows the basic header layout in relation to IP and the transport protocol used. For details regarding the socket handling see AUTOSAR Socket Adaptor SWS.

2.3.1.1 Mapping of IP Addresses and Ports

For the response and error message the IP addresses and port number of the transport protocol must match the request message. This means:

- Source IP address of response = destination IP address of request.
- Destination IP address of response = source IP address of request.
- Source port of response = destination port of request.
- Destination port of response = source port of request.
- The transport protocol (TCP or UDP) stays the same.

2.3.2 Message ID [32 Bit]

The Message ID is a 32 Bit identifier that is used to dispatch the RPC call to method of an application and to identify a notification. The Message ID has to uniquely identify a method.

The assignment of the Message ID is up to the user; however, the Message ID has to be unique for the whole system (i.e. the vehicle). The Message ID can be best compared to CAN IDs and should be handled with a comparable process. The next section describes how structure the Message IDs in order to ease the organization of Message IDs.

2.3.2.1 Recommended Structure of the Message ID

In order to structure the different methods and events, they are clustered into services. Services have a set of methods and events as well as a Service ID, which is only used for this service. The events may in addition be clustered into event groups, which simplify the registration of events.

For RPC calls we structure the ID in 2^x services with $2^{(32-x)}$ methods:

Service ID [x bits]	0 (single bit)	Method ID [32-(x+1) bits]

It is required that x is unique in the whole system. We currently recommend the use of x=16, which allows for up to 65536 services with up to 65536 methods each.

Since events (see Notification or Publish/Subscribe) are transported using RPC, the ID space for the events is further structured:

Service ID [x bits]	1 (single bit)	Event Group ID [y bits]	Event ID [32-(x+y+1) bits]

The size of y is determined individually for each service and their events.

The combination of Event Group ID and Event ID together are known as Notification ID.

2.3.3 Length [32 Bit]

Length is a field of 32 Bits containing the length in Byte of the payload beginning with the Request ID/Client ID. The Length does not cover the portion of the header including the Message ID and the Length field since it is based on capabilities of the AUTOSAR Socket Adapter.

2.3.4 Request ID [32 Bit]

The Request ID allows a client to differentiate multiple calls to the same method. Therefore, the Request ID has to be unique for a single client and server combination only. When generating a response message, the server has copy the Request ID from the request to the response message. This allows the client to map a response to the issued request even with more than one request outstanding.

Request IDs might be reused as soon as the response arrived or is not expected to arrive anymore (timeout). In most automotive use cases a very low number of outstanding requests are expected. For small systems without the possibility of parallel requests, the Request ID might always set to the same value.

For AUTOSAR systems the Request ID needs to be structured as shown in the next section. Even for non-AUTOSAR systems it is recommended to encode the callers Client ID as shown in the next section.

2.3.4.1 AUTOSAR-specifics

In AUTOSAR the Request ID is constructed of the Client ID and Session ID:

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

The Client ID is the unique identifier for the calling client inside the ECU. The Session ID is a unique identifier chosen by the client for each call. If session handling is not used, the Session ID must be set to 0x0000.

2.3.5 Protocol Version [8 Bit]

Protocol Version is an 8 Bit field containing the SOME/IP protocol version, which currently set to 0x01.

2.3.6 Interface Major Version [8 Bit]

Interface Major Version is an 8 Bit field that contains the Major Version of the Service Interface.

Rationale: This is required to catch mismatches in Service definitions and allows debugging tools to identify the Service Interface used, if version is used.

2.3.7 Message Type [8 Bit]

The Message Type field is used to differentiate different types of messages and may contain the following values:

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 re-sponse*
0x40	*REQUEST ACK*	*Acknowledgment for REQUEST (optional)*
0x41	*REQUEST_NO_RETURN ACK*	*Acknowledgment for REQUEST_NO_RETURN (informational)*
0x42	*NOTIFICATION ACK*	*Acknowledgment for NOTIFICATION (informational)*
0x80	*RESPONSE*	*The response message*
0x81	*ERROR*	*The response containing an error*
0xC0	*RESPONSE ACK*	*Acknowledgment for RESPONSE (informational)*

0xC1	*ERROR ACK*	*Acknowledgment for ERROR (informational)*

Regular request (message type 0x00) will be answered by a response (message type 0x80), when no error occurred. If errors occur an error message (message type 0x81) will be sent. It is also possible to send a request that does not have a response message (message type 0x01). For updating values through notification a callback interface exists (message type 0x02).

For all Messages an optional acknowledgment (ACK) exists. These can be used in cases that the transport protocol (i.e. UDP) does not acknowledge a received message. ACKs are only transported when the interface specification requires it. Only the usage of the REQUEST_ACK is currently specified in this document and shall be implemented. All other ACKs are currently informational and do not need to be implemented.

2.3.8 Return Code [8 Bit]

The Return Code is used to signal whether a request was successfully been processed. For simplification of the header layout, every message transports the field Return Code. Messages of Type REQUEST, REQUEST_NO_RETURN, and Notification have to set the Return Code to 0x00. The allowed Return Codes are:

Message Type	***Allowed Return Codes***
REQUEST	*N/A set to 0x00*
REQUEST_NO_RETURN	*N/A set to 0x00*
NOTIFICATION	*N/A set to 0x00*
RESPONSE	*Set to Error-Code. 0x00 is reserved for “no error”*
ERROR	*0x01 (to be synchronized with AUTOSAR)*

2.3.9 Payload [variable size]

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

The size of the payload field depends on the transport protocol used. With UDP the payload can 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.

2.4 Serialization of Parameters and Data Structures

The serialization is based on the parameter list defined by the interface specification. To allow migration of the service interface the deserialization code shall ignore parameters attached to the end of previously known parameter list; i.e. parameters that were not defined in the interface specification used to generate or parameterize the deserialization code.

The interface specification defines the exact position of all parameters in the PDU and has to consider the memory alignment. The serialization must not try to automatically align parameters but shall be aligned as specified in the interface specification. SOME/IP payload should be placed in memory so that the SOME/IP payload is suitable aligned. For infotainment ECUs an alignment of 8 Bytes (i.e. 64 bits) should be achieved, with AUTOSAR ECUs at least an alignment of 4 Bytes must be achieved.

In the following the deserialization of different parameters is specified.

2.4.1 Basic Datatypes

The following basic datatypes are supported:

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	
sint8	signed Integer	8	
sint16	signed Integer	16	
sint32	signed Integer	32	
float32	floating point number	32	IEEE 754 binary32 (Single Precision)
float64	floating point number	64	IEEE 754 binary64 (Double Precision)

The Byte Order is specified for each parameter by the interface definition.

2.4.1.1 AUTOSAR Specifics

See AUTOSAR SWS COM 7.2.2 (COM675) for supported data types.

AUTOSAR COM module shall support endianess conversion for all Integer types (COM007).

AUTOSAR defines boolean as the shortest supported unsigned Integer (Platform Types PLATFORM027). SOME/IP uses 8 Bits.

2.4.2 Structured Datatypes (structs)

The serialization of a struct is based on its in-memory layout. Especially for structs it is important to consider the correct memory alignment. Insert dummy/padding elements in the interface definition if needed for alignment.

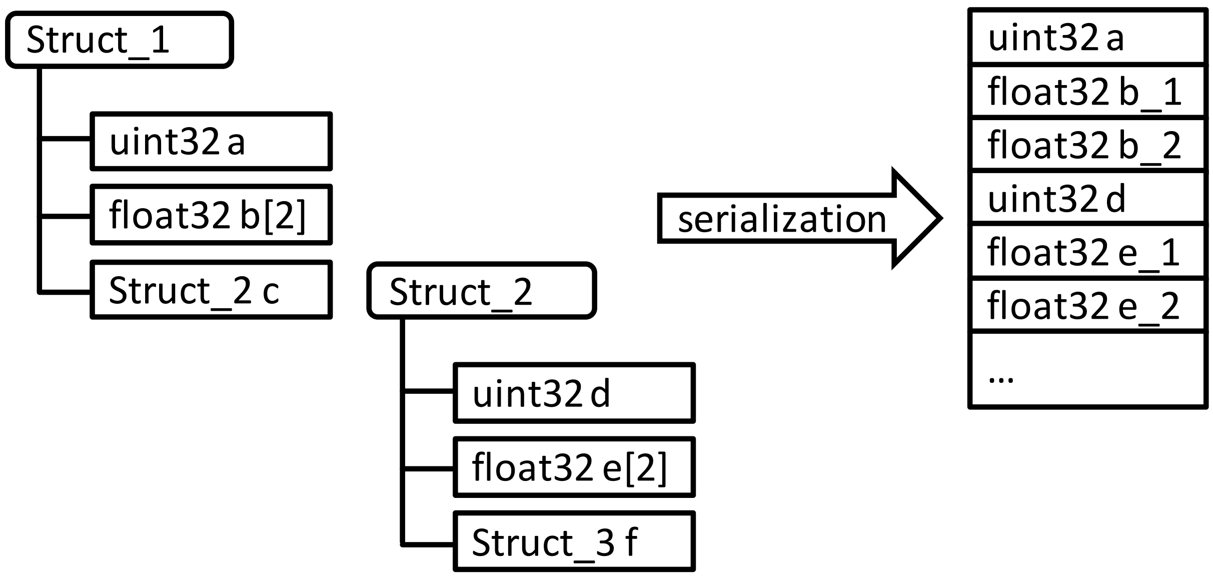

Figure 2 Serialization of Structs

2.4.3 Strings (fixed length)

Strings are encoded using Unicode and are terminated with a “\0”-character. The length of the string (this includes the “\0”) in Bytes has to be specified in the interface definition. Fill unused space using “\0”.

Different Unicode encoding may be used including UTF-8, UTF-16BE, and UTF-16LE. Since these encoding have a dynamic length of bytes per character, the maximum length in bytes is up to three times the length of characters in UTF-8 plus 1 Byte for the termination with a “\0” or two times the length of the characters in UTF-16 plus 2 Bytes for a “\0”.

The chosen String encoding shall be specified in the interface definition.

2.4.4 Strings (dynamic length)

Strings with dynamic length start with a length field [32 Bit]. The length is measured in Bytes and is followed by the "\0"-terminated string data. The interface definition must also define the maximum number of bytes the string (including termination with "\0") can occupy.

Supported encodings are defined as in Section 2.4.3.

If the interface definition hints the alignment of the next data element the string shall be extended with "\0" characters to meet the alignment.

2.4.5 Arrays (fixed length)

The length of fixed length arrays is defined by the interface definition. They can be seen as repeated elements. In Section 2.4.7 dynamic length arrays are shown, which can be also used. However fixed length arrays can easier be integrated into early version of AUTOSAR and very small devices; thus, both options are being supported.

2.4.5.1 One-dimensional

The one-dimensional arrays with fixed length *n* carry exactly *n* elements of the same type. The layout is shown in Figure 3.

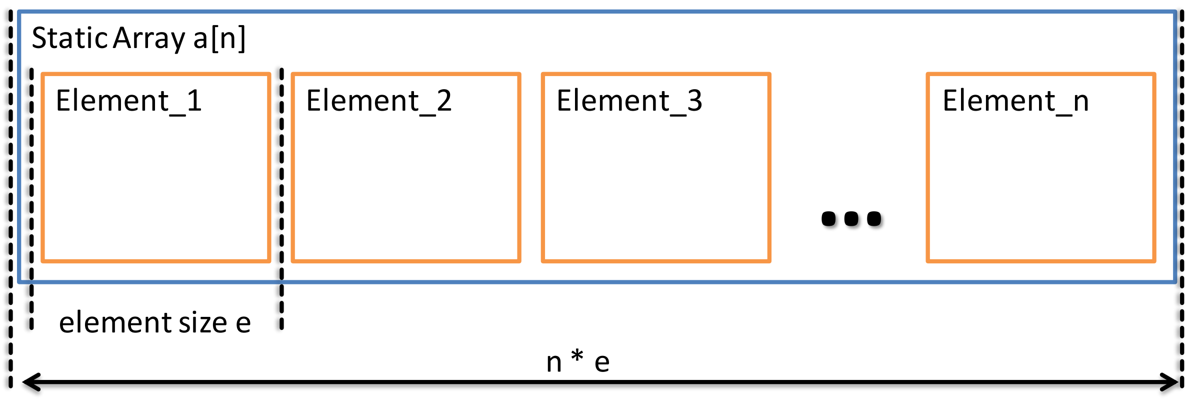

Figure 3 One-dimensional array (fixed length)

2.4.5.2 Multidimensional

The serialization of multidimensional arrays follows the in-memory layout of multidimensional arrays in the C++ programming language (row-major order) and is shown in Figure 4.

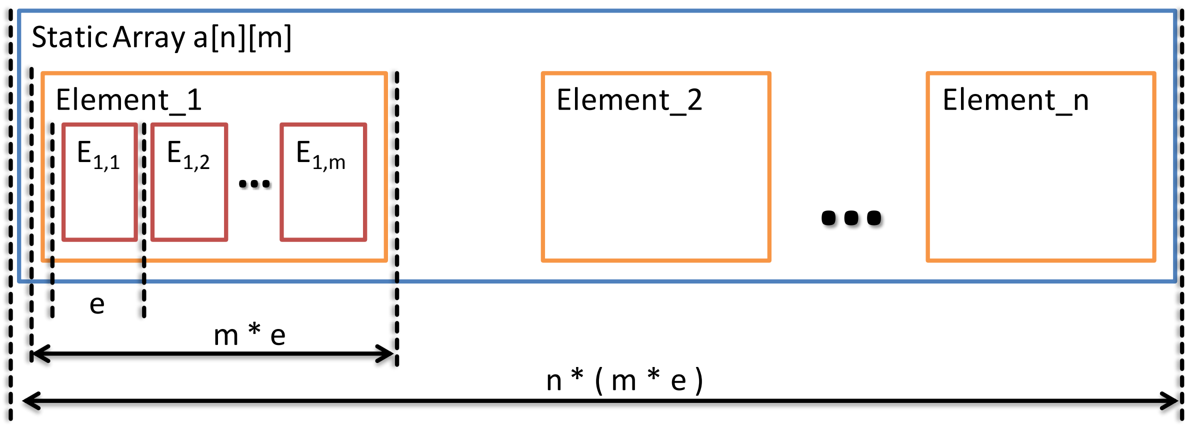

Figure 4 Multidimensional array (fixed length)

2.4.5.3 AUTOSAR Specifics

Consult AUTOSAR SWS RTE chapter 5.3.4.4 for Arrays.

As of today only a single uint8 array is supported as dynamic data structure.

2.4.6 Optional Fields

Optional Fields shall be encoded as array with 0 to 1 elements. For the serialization of arrays with dynamic length see Section 2.4.7.

2.4.7 Dynamic Length Arrays

The layout of arrays with dynamic length basically is based on the layout of fixed length arrays. To determine the size of the array the serialization adds a length field (32 bit) in front of the data, which counts the bytes of the array. The length does not include the size of the length field. Thus, when transporting an array with zero elements the length is set to zero.

The layout of dynamic arrays is shown in Figure 5 and Figure 6.

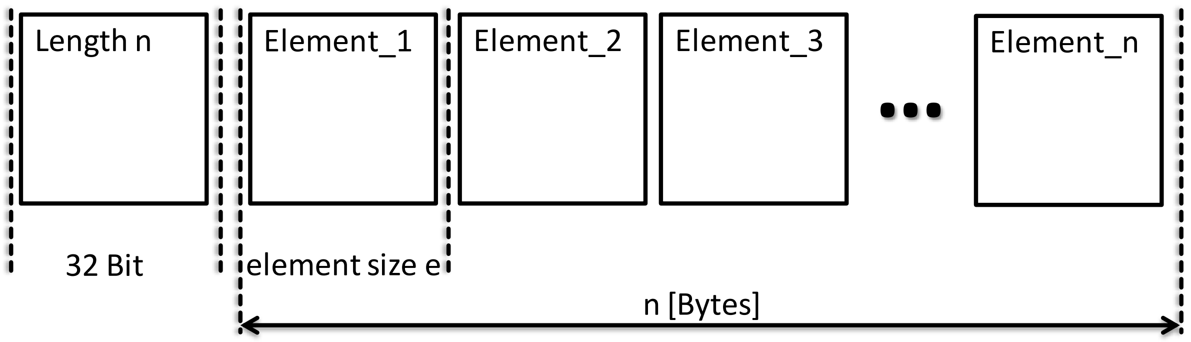

Figure 5 One-dimensional array (dynamic length)

In the one-dimensional array one length field is used, which carries the number of bytes used for the array. The number of elements can be easily calculated by dividing by the size of an element.

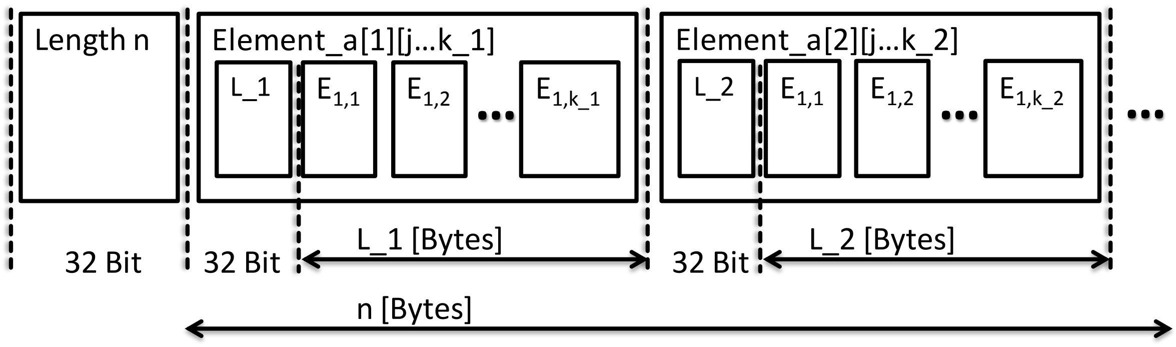

Figure 6 Multidimensional array (dynamic length)

In multidimensional arrays multiple length fields are needed.

The interface definition shall define the maximal length of each dimension in order to allow for static buffer size allocation.

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

2.4.8 Union / Variant

A union (also called variant) is a parameter that can contain different types of elements. For example, if one defines a union of type uint8 and type uint16, the union may carry an element of uint8 or uint16. It is clear that that when using different types of elements the alignment of subsequent parameters may be distorted. To resolve this, padding might be needed.

The serialization layout of unions in SOME/IP is as follows:

Length field [32 bit]
Type field [32 bit]
Element including padding [sizeof(padding) = length – sizeof(element)]

The length field defines the size element and padding in bytes and does not include the size of the length field and type field.

The type field describes the type of the element. Possible values of the type field are defined by the interface specification for each union separately. The types are encoded as in the interface specification in ascending order starting with 1. The 0 is reserved for the NULL type – i.e. an empty union. The usage of NULL must be allowed by the Interface Definition.

The element is serialized depending on the type in the type field. In conjunction with the length field padding can be added behind the element. The deserializer must skip bytes according to the length field. The value of the length field for each type must be defined by the interface specification.

By using a struct different padding layouts can be achieved.

2.4.8.1 Example: Union of uint8/uint16 both padded to 32 bit

In the interface specification a union of uint8 and uint16 is specified. Both are padded to the 32 bit boundary (length=4).

A uint8 will be serialized like this:

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

A uint16 will be serialized like this:

Length = 4 Bytes		
Type = 1		
uint16	Padding 0x00	Padding 0x00

2.5 Map / Dictionary

Maps or dictionaries can be easily been described as an array of key-value-pairs. The most basic way to implement a map or dictionary would be an array of a struct with two fields: key and value. Since the struct has no length field, this is as efficient as a special map or dictionary type could be. When choosing key and value as uint16, a serialized map with 3 entries looks like this:

Length = 12 Bytes	
key0	value0
key1	value1
key2	value2

3 RPC Protocol specification

This chapter describes the RPC protocol of SOME/IP.

3.1 Transport Protocol Bindings

In order to transport SOME/IP messages of IP different transport protocols may be used. SOME/IP currently support UDP and TCP. Their bindings are explained in the following sections, while Section 4.1 discusses which transport protocol to choose.

3.1.1 UDP Binding

The UDP binding of SOME/IP is straight forward by transporting one SOME/IP message per UDP packet. The SOME/IP messages must not be fragmented. Therefore care must be taken that SOME/IP messages are not too big, i.e. up to 1400 Bytes of SOME/IP payload. Messages with bigger payload may be discarded.

The header format allows transporting more than one SOME/IP message in a single UDP packet. The SOME/IP implementation can easily identify the end of a SOME/IP message by means of the SOME/IP length field. Based on the UDP lengths field SOME/IP can determine if there are additional SOME/IP messages in the UDP packet.

As optimization the UDP binding of SOME/IP can use acknowledgment messages especially for request/response communication that triggers a long running operation at server side that must be completed before sending a result (transport or processing acknowledgement). The acknowledgment messages are SOME/IP messages with exactly the same header fields but with the changed message type and without a payload. The use of these additional acknowledgment messages must be configured by the interface specification.

3.1.1.1 AUTOSAR specific

Based on the Socket Adaptor concept AUTOSAR can divide an incoming UDP packet into different i-PDUs. However, AUTOSAR is currently not able to combine different i-PDUs and send an UDP-Packet with more than one SOME/IP message.

3.1.2 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 the transport of a large number of SOME/IP messages after each other (pipelining).

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

When the TCP connection is lost, outstanding request should be handled as timeouts. Since TCP handles reliability, additional means of reliability are not needed. Error handling is discussed in detail in Section 3.5.

3.2 Request/Response Communication

One of the most common communication patterns is the request/response pattern. One party (in the following called the client) sends a request message, which is answered by another party (the server).

In the SOME/IP header the request message the client has to do the following:

- 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 second part of the SOME/IP header) + length of the serialized payload
- Optionally set the Request ID to a unique number (must be unique for client only)
- Set the Protocol Version according Section 2.3.5
- Set the Interface Version according to the interface definition
- Set the Message Type to Request (i.e. 0x00)

- Set the Return Code to 0x00

The server builds it header based on the header of the client and does in addition:

- Construct the payload
- Set the length to the 8 Bytes + new payload size
- Set the Message Type to RESPONSE (i.e. 0x80) or ERROR (i.e. 0x81)

3.2.1 AUTOSAR Specific

AUTOSAR should implement Request-Response by means of the Client/Server-Functionality. For intermediate implementations it might be necessary to implement the inter-ECU communication by means of Sender/Receiver-Functionality. In this case the semantics and syntax of the PDU must not be different to this specification.

3.3 Fire&Forget Communication

Requests without Response message are called Fire&Forget. The implementation is basically the same as for Request/Response with the following differences:

- There is no response message.
- The message type is set to REQUEST_NO_RETURN (i.e. 0x01)

Fire&Forget messages do no return an error. Error handling and return codes must be implemented by the application when needed.

3.3.1 AUTOSAR Specific

Fire&Forget should be implemented using the Sender/Receiver-Functionality.

3.4 Notification

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

The RPC of SOME/IP is used only for transporting the updated value and not for the publishing and subscription mechanisms. These mechanisms are explained in Section 3.4.2.

When more than one subscribed client on the same ECU exists, the system should handle the replication of notifications in order to save transmissions on the bus. This is especially important, when notifications are transported using multicast messages.

3.4.1 Strategy for sending notifications

For different use cases different strategies for sending notifications are possible and must defined in the service interface. The following examples are common:

- Cyclic update – send an updated value in a fixed interval (e.g. every 10 ms)
- 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.

3.4.2 Publish/Subscribe Handling

Publish/Subscribe handling shall be implemented according to Lastenheft Ethernet Configuration [1].

3.4.3 AUTOSAR Specific

Notifications are transported with AUTOSAR Sender/Receiver-Functionality. In case of different notification receivers within an ECU, the replication of notification messages is done in the RTE. This means a notification message must be only sent once per ECU.

3.5 Error Handling

One can partition the error handling into application and communication errors.

3.5.1 Transporting Application Error Codes and Exceptions

For the error handling two different mechanisms are supported. All messages have a return code field to carry the return code. However, only responses (Message Types 0x80 and 0x81) use this field to carry a return code to the request (Message Type 0x00) they answer. All other messages set this field to 0x00 (see Section 2.3.7). For more detailed errors the layout of the Error Message (Message Type 0x81) can carry specific fields for error handling, e.g. an Exception String. Error Messages are sent instead of Response Messages.

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.

3.5.2 Return Code

The Error Handling is based on an 8 Bit Std_returnType of AUTOSAR. The two most significant bits are reserved and must be set to 0. The receiver of a return code must ignore the values of the two most significant bits.

The following Return Codes are currently defined and shall be implemented as described:

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.
0x06	E_TIMEOUT	A timeout occurred.
0x07	E_WRONG_PROTOCOL_VERSION	Version of SOME/IP protocol not supported
0x08	E_WRONG_INTERFACE_VERSION	Interface version mismatch
0x09 - 0x1f	RESERVED	Reserved for generic SOME/IP errors. These errors will be specified in future versions of this document.
0x20 - 0x3f	RESERVED	Reserved for specific errors of services and methods. These errors are specified by the interface specification.

3.5.3 Error Message Format

For a more flexible error handling, SOME/IP allows the user to specify a message layout specific for errors instead of using the message layout for response messages. This is defined by the API specification and can be used to transport exceptions of higher level programming languages.

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.

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.

3.5.3.1 AUTOSAR Specific

In AUTOSAR this feature can be supported by means of a PDU Multiplexer, which is already used for similar applications today.

3.5.4 Communication Errors and Handling of Communication Errors

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

- *Maybe* – the message might have reached the communication partner
- *At least once* – the message reached at least once the communication partner
- *Exactly once* – the message reached exactly once the communication partner

When using these 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 7 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 must signal E_TIMEOUT to the client.

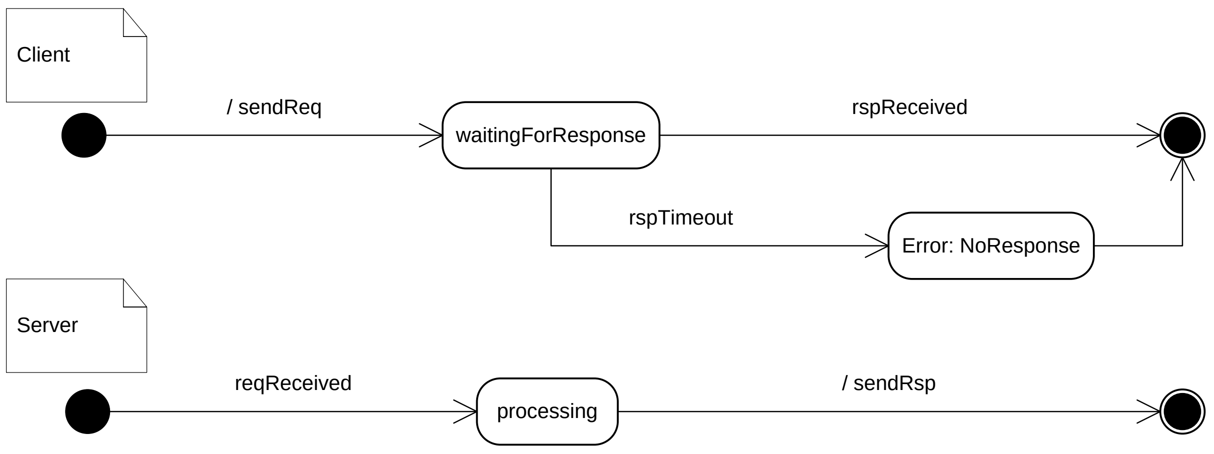

Figure 7 State Machines for Reliability "Maybe"

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

Additional mechanisms to reach higher reliability may be implemented in the application or in a SOME/IP implementation. Keep in mind that the communication does not have to implement these features. Section 3.5.4.1 describes such optional reliability mechanisms.

3.5.4.1 Application based Error Handling

The application can easily implement “at least once” reliability by using idempotent operations (i.e. operation that can be executed multiple times without side effects) and using a simple timeout mechanism. Figure 8 show the state machines for “at least once” reliability using implicit acknowledgments. When the client sends out the request it starts a timer with the timeout specified for the specific method. If no response is received before the timer expires (round transition at the top), the client will retry the operation. Typical number of retries would be 2, so that 3 requests are sent.

The number of retries, the timeout values, and the timeout behavior (constant or exponential back off) are outside of the SOME/IP specification and may be added to the interface definition.

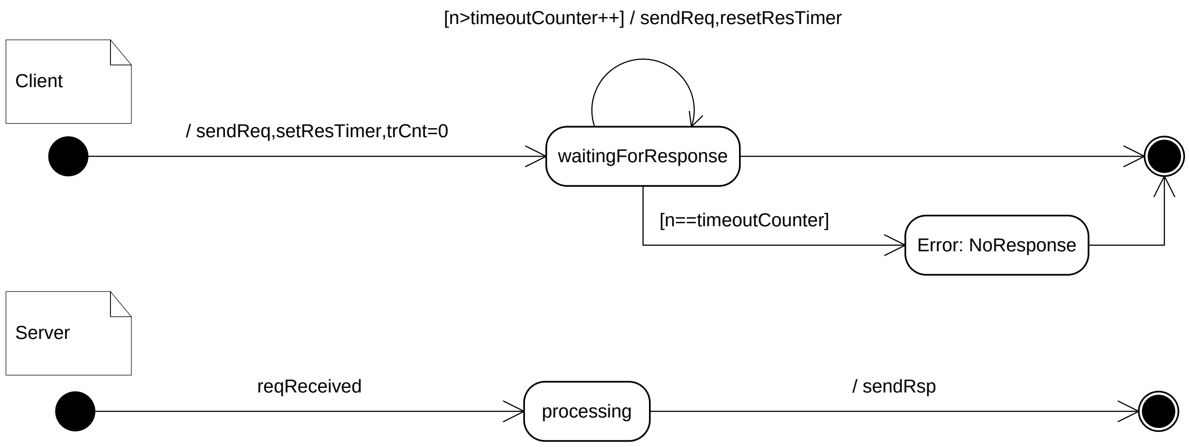

Figure 8 State Machines for Reliability "At least once" (idempotent operations)

3.6 Multiple Service-Instances

While different services can share the same port number, multiple Service Instances on a single ECU must listen on different ports per instance. Instances on different ECUs are identified through different Instance-IDs. Those are used for Service Discovery but are not contained in the RPC 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 should use exactly TCP port x for its services.

3.7 Service Discovery

Service Discovery is based on Bonjour. The exact specification can be found in Lastenheft Ethernet Configuration [1].

4 Guidelines (informational)

4.1 Choosing the transport protocol

SOME/IP directly support the two most used transport protocols of the Internet: 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 can not only handle bit errors but also segmentation, loss, duplication, reordering, and network congestion; thus, TCP is the more powerful transport protocol.

For use inside the vehicle, requirements are not the same as for the Internet. For many applications, we require a very short timeouts to react in a very short time. 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; thus, only being able to transport smaller chunks of data.

Guideline:

- Use UDP if very hard latency requirements (<100ms) in case of errors is needed
- Use TCP only if very large chunks of data need to be transported (>> 1kb) and hard latency requirements in the case of errors exists
- Try using external transport (Network File System, APIX link, 1722, …) when more suited for the use case. Just transport file handle or similar. This gives the designer more freedom (caching etc).

The choice of the transport protocol must be documented for every method in the interface specification. Methods should not support the usage of more than one transport protocol.

4.2 Implementing Advanced Features in AUTOSAR Applications

Beginning with AUTOSAR 4.0.3 SOME/IP will be supported. Unfortunately, not all features of SOME/IP can be directly supported within AUTOSAR (e.g. dynamic length arrays). In the uncommon case that an advanced feature is needed within an AUTOSAR implementation not supporting it directly, a solution exists: The advanced feature shall be implemented inside the application by passing the SOME/IP payload or parts of it by means of a uint8 buffer through AUTOSAR. For AUTOSAR the fields seem to be just a dynamic length uint8 array and must be configured accordingly.

Keep in mind that with AUTOSAR 4.0.3 it is only possible to have a single uint8 array and it must be at the end of the payload.

4.3 Serialization of Data Structures Containing Pointer

For the serialization of data structures containing pointers (e.g. a tree in memory), the pointers cannot be just transferred using a data type (e.g. uint8) but must be converted for transport. Different approaches for the serialization of pointers exist. We recommend the following approaches.

4.3.1 Array of data structures with implicit ID

When transporting a set of data structures with pointers that is small enough to fit into a single RPC message:

- Store data structures (e.g. tree nodes) in array
- Use position in array as ID of stored data structure
- Replace pointers with IDs of the data structures pointed to

4.3.2 Array of data structures with explicit ID

With larger sets of data structures additional problems have to be resolved. Since not all data structures fit into a single message the IDs have to be unique over different messages. This can be achieved in different ways:

- Add an offset field to every message. The ID of an array element will be calculated by adding the offset to its position in the array. Keep in mind that the offset needs to be carefully been chosen. If for example every message can contain up to ten data structures (0-9), the offset could be chosen as 0, 10, 20, 30, and so on.
- Store an explicit ID by using an array of structs. The first field in the struct will be an ID (e.g. uint32) and the second field the data structure itself. For security and reliability reasons the pointer (i.e. the memory address) should never be used directly as ID.

5 Compatibility rules for Interface Design (informational)

As for all serialization formats migration towards a newer service interface is somewhat limited. Using a set of compatibility rules, SOME/IP allows for evolution of service interfaces. One can do the following additions and enhancements in a non-breaking way:

- Add new method to a service
  - Must be implemented at server first.
- Add parameter to the end of a method's in or out parameters
  - When the receiver adds them first, a default value has to be defined
  - When the sender adds them first, the receiver will ignore them
- Add an exception to the list of exceptions a method can throw
  - Should update client first
  - If exception is unknown, "unknown exception" needs to thrown. The exception description string however can be copied over.
- Add new type to union
  - Should update receiver first
  - Can be skipped if unknown
- Define a new data type for new methods
- Define a new exception for new methods

Not of all these changes can be introduced at the client or server side first. In some cases only the client or server can be changed first. For example, when sending an additional parameter with a newer implementation, the older implementation can only skip this parameter.

When the receiver of a message adds for example a new parameter to be received, it has to define a default value. This is needed in the case of an older sender sending the message without the additional parameter.

Some changes in the interface specification can be implemented in a non-breaking way:

- Delete Parameters in Functions
  - Need to add default value first at receiver first and parameters need to be at end of list
- Remove Exceptions from functions
  - Trivial at server side
  - Client needs to throw "unknown exception", if encountering old exception
- Renaming parameters, functions, and services is possible since the names are not transmitted. The generated code only looks at the IDs and the ordering of parameters, which must not be changed in migration.

Not currently supported are the following changes:

- Reordering parameters
- Adding / deleting fields to/from structs
- Replace supertype by subtype (as in object oriented programming languages)

6 Transporting CAN and FlexRay Frames

SOME/IP should not be used to simply transport CAN or FlexRay frames. However, the Message ID space needs to be coordinated between both use cases.

The following layout of a message could be used for CAN or FlexRay Frame transport:

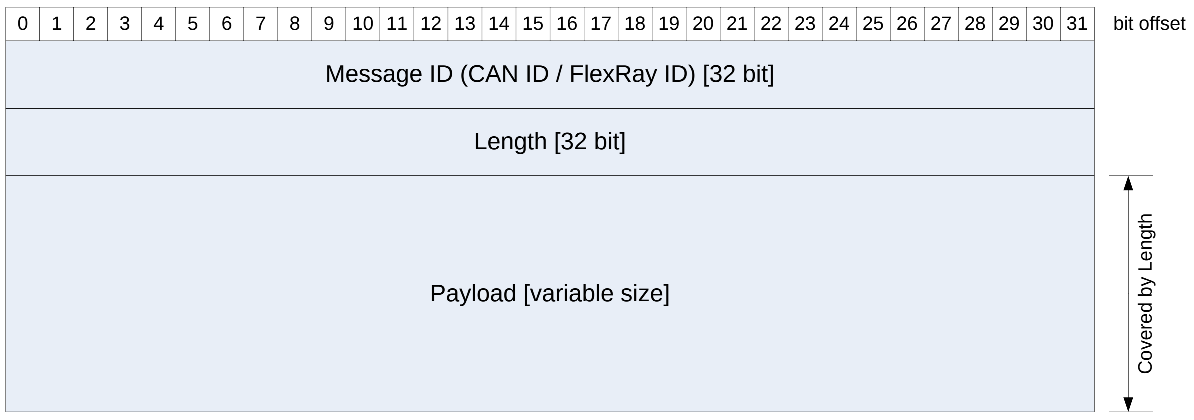

Figure 9 Format of CAN or FlexRay frame transport

The AUTOSAR Socket-Adapter uses the Message ID and Length to construct the needed internal PDUs but does not look at other fields. Therefore, one has to encode the CAN ID (11 or 29 bits) or the FlexRay ID (6+6+11 bits) into the Message ID field. The ID must be aligned to the least significant bit of the Message ID and the unused bits shall be set to 0. An 11 bit CAN identifier would be therefore transported in the bit position 21 to 31.

Since this potentially could require a big portion of the Message ID space, it is recommended to bind SOME/IP and CAN/FlexRay transports to different transport protocol ports, so that different ID spaces for the Message IDs exist.

Keep in mind that when transporting a CAN frame of 8 Byte over Ethernet an overhead of up to 100 Bytes might be needed in the near future (using IPv6 and/or security mechanisms). So it is recommended to use larger RPC calls as shown in the first part of the document instead of small CAN like communication.

6.1 AUTOSAR specific

AUTOSAR currently does not allow for sending more than one CAN or FlexRay frame inside an IP packet. AUTOSAR however allows receiving more than one CAN or FlexRay frame inside an IP packet by use of the length field.

7 Integration of SOME/IP in AUTOSAR (informational)

In this chapter discusses the dependencies and processes in order to use SOME/IP in AUTOSAR.

Figure 10 shows an example for an AUTOSAR tool chain. The interface definition is specified in FIBEX 4 and must be added to the AUTOSAR System Template by means of a generator or converter. Using the AUTOSAR System Template different configurations and mappings are generated (e.g. Socket Adaptor (SoAd) configuration, all configurations for Client/Server- and Sender/Receiver-Mappings, and PDU multiplexers).

When using features not supported by the currently used AUTOSAR, it might be necessary to add another generator to generate the needed stub and skeleton code for clients and servers.

7.1 Rationale

FIBEX 4 [2] was already extended to specifying the needed information:

- Ethernet topology and configuration (FIBEX4Ethernet)
- Protocol configuration (FIBEX4IT)
- Services and Methods (FIBEX4Services

Currently no alternative exists. While for AUTOSAR the System Template will be enhanced, the AUTOSAR System Template might be a good solution in the future, other platforms might have legal or organizational difficulties of using it.

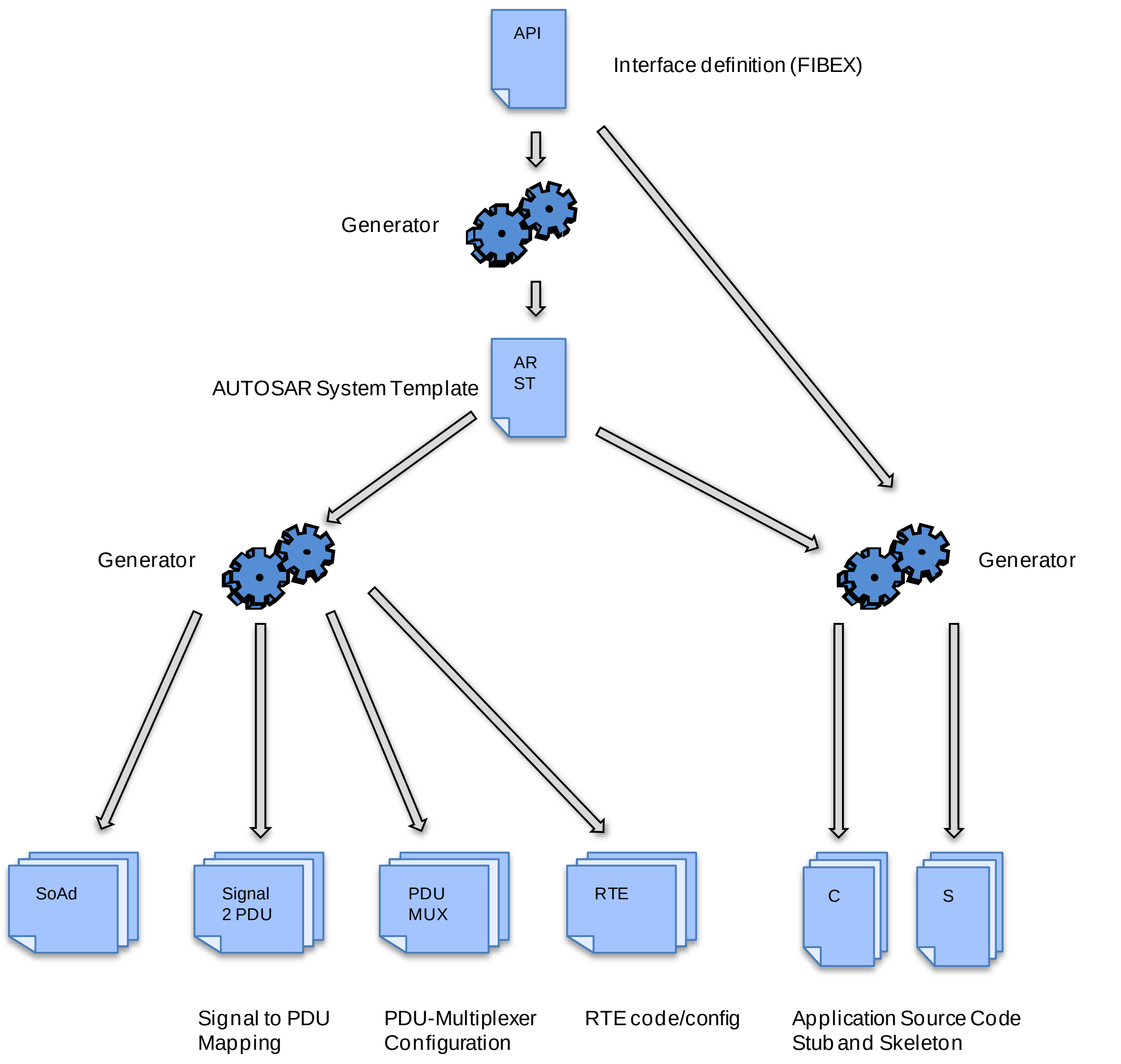

Figure 10 AUTOSAR tool chain (Example)

8 References

[1] **BMW GROUP.** *ETHERNET CONFIGURATION.* [LH 10216504].

[2] **ASAM.** *Data Model for ECU Network Systems (Field Bus Data Exchange Format).* 4.0.0.

[3] **AUTOSAR.** *Specification of Socket Adaptor (SWS).* 1.1.0.

[4] **AUTOSAR.** *Specification of Standard Types (SWS).* 1.3.0.

[5] **AUTOSAR.** *Specification of RTE (SWS).* 3.1.0.