VECTOR

MICROSAR Classic Sd

Technical Reference

Service Discovery

Version 13.1.0

Authors	vispcn, viskjs, visfaz
Status	Released

Document Information

History

Author	Date	Version	Remarks
vispcn	04.07.2014	1.0.0	Creation of document
vispcn	14.08.2014	1.0.x	Improvement of documentation
vispcn	02.12.2014	1.1.x	Added new DEM and DET errors introduced in AUTOSAR v4.2.1
vispcn	28.01.2015	2.0.x	Changed configuration according to AUTOSAR v4.2.1
vispcn	13.04.2015	2.1.x	Updated DEM configuration
vispcn	03.09.2015	3.0.x	Support of post-build loadable
vispcn	02.06.2016	4.0.x	Added user callouts to handle configuration options
vispcn	24.02.2017	5.0.x	Updated Service and Error IDs
vispcn	29.08.2017	5.1.x	Updated Service and Error IDs according AUTOSAR.
vispcn	04.01.2018	6.0.x	Described possibility to share consumed eventgroup RX multicast connections.
vispcn	20.02.2018	6.1.x	Added Sd_Cbk.h as static file instead of generated file and updated constraints for module initialization.
vispcn	01.07.2019	7.0.0	Added description of retry subscription mechanism for requested eventgroups.
vispcn	09.08.2019	7.0.1	Added constraints for main function preemption.
viskjs	15.05.2020	7.1.0	Added description of Rx Queue process limit.
viskjs	03.12.2020	7.2.0	Added description of SdServiceGroups.
viskjs	09.03.2021	7.3.0	Added description of version driven find behavior, auto available/required and max remote clients. Updated screenshots.
visfaz, viskjs	23.03.2021	7.4.0	Updated document to new template. Measurement Data API.
viskjs	15.04.2021	7.5.0	Added description of Rtm Measurement configuration and new parameter SdInstanceLocalAdressCheckLength.
visfaz	29.04.2021	7.6.0	Reporting of runtime errors can be disabled.
visfaz	22.06.2021	8.0.0	Added call to new SoAd API SoAd_ForceReleaseRemoteAddr.
visfaz	07.07.2021	8.1.0	Support of post-build selectable.
visfaz	06.10.2021	9.0.0	Refined description of post-build loadable and selectable features.

viskjs	31.01.2022	10.0.0	Added Sd_MemMap.h to dynamic files of Embedded Implementation.
visfaz	16.01.2023	11.0.0	Dynamic update of SD Instance ID.
visfaz	28.03.2023	12.0.0	Added description of multi-partition support.
visfaz	17.05.2023	12.1.0	Updated description of multi-partition support after implementation of data separation.
viskjs	14.11.2023	13.0.0	Added note about nested critical sections.
viskjs	24.01.2024	13.1.0	Support of subscription to multicast address pre-defined by ClientService.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of Service Discovery	R4.2.2
[2]	AUTOSAR	List of Basic Software Modules	R1.7.0
[3]	AUTOSAR	Specification of Socket Adaptor	R2.2.0
[4]	AUTOSAR	Specification of Basic Software Mode Manager	R1.3.0
[5]	AUTOSAR	Specification of Development Error Tracer	R4.1.2
[6]	AUTOSAR	Specification of Diagnostic Event Manager	R4.1.2
[7]	Vector	User Manual Post-Build Loadable	see delivery
[8]	Vector	User Manual Identity Manager	see delivery
[9]	Vector	Technical Reference MemMap	see delivery

Scope of the Document

This technical reference describes the general use of the Service Discovery (Sd) basis software.

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module SD as specified in [1].

Supported AUTOSAR Release:	4.2.2	
Supported Configuration Variants:	pre-compile, post-build	
Vendor ID:	SD_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	SD_MODULE_ID	171 decimal (according to ref. see [2])

The AUTOSAR Service Discovery (SD) module offers functionality to detect and offer available services within the vehicle network in order to implement service-oriented communication. Therefore, a SD server publishes its available services via IP multicast messages in the network. A SD client which receives this offer message and is interested in the service can use the functionality provided by the server.

The Service Discovery module is located between the AUTOSAR BSW Mode Manager module (BswM see [4]) and the AUTOSAR Socket Adaptor module (SoAd see [3]).

1.1 Architecture Overview

The following figure shows where the SD is located in the AUTOSAR architecture.

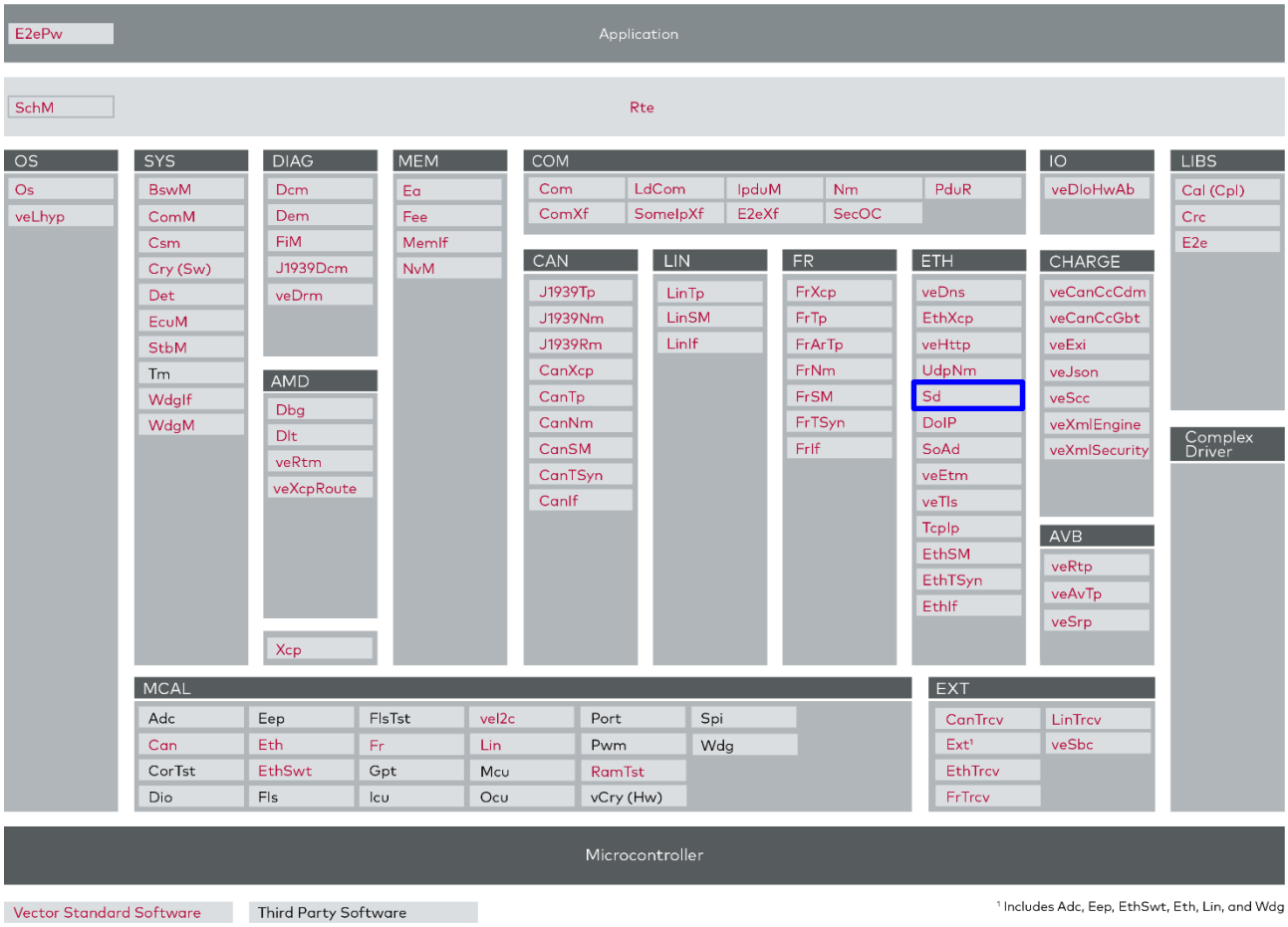

Figure 1-1 AUTOSAR 4.2 Architecture Overview

The next figure shows the interfaces to adjacent modules of the SD. These interfaces are described in chapter 4.

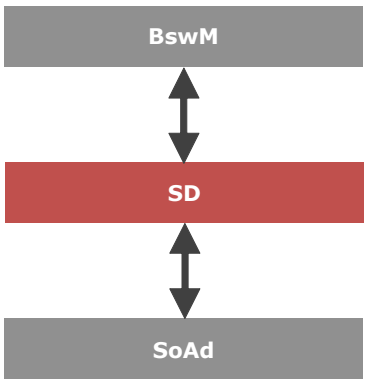

Figure 1-2 Interfaces to adjacent modules of the SD

Applications do not access the services of the BSW modules directly. They use the service ports provided by the BSW modules via the RTE. The Service Discovery does not support any service ports.

2 Functional Description

2.1 Features

The features listed in the following tables cover the complete functionality specified for the SD.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 2-1 Supported AUTOSAR standard conform features

> Table 2-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further SD functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 2-3 Features provided beyond the referenced AUTOSAR Standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Realization of client and server services in parallel
Server service features
Handling of provided methods and eventgroups
Dynamic IP address handling of multiple clients
Switch between unicast and multicast transmission based on the number of subscribed clients
Client service features
Handling of consumed methods and eventgroups
Dynamic configuration of IP addresses during runtime
Reporting of current state to BswM
Dynamic state handling by BswM during runtime
Differentiation of received messages between unicast and multicast
Response timing adapted to receive path (unicast, multicast)
Handling of UDP and TCP connections
Generation of configuration options based on configured host name and capability records
Use the IP address specified in a SD Endpoint Option in order to reply to messages
Differentiate multiple subscriptions to the same eventgroup from the same client by the counter field introduced in the eventgroup entries
Support of post-build loadable
Support of post-build selectable

Table 2-1 Supported AUTOSAR standard conform features

2.1.1 Deviations

The following features specified in [1] are not supported:

Not Supported AUTOSAR Standard Conform Features
No deviations known.

Table 2-2 Not supported AUTOSAR standard conform features

2.1.2 Additions/ Extensions

The following features are provided beyond the referenced AUTOSAR Standard Release Version:

Features Provided Beyond The AUTOSAR Standard
SD module receives information about the current state of the TCP connection established by a SdClientService. The information is provided by the SOAD API `Sd_SoConModeChg()`. This mechanism allows it to trigger the subscription only if the TCP connection is successfully established. Additionally, the SD also detects lost connections and can initiate a reconnection.
The additional SoAd API (`SoAd_GetRcvRemoteAddr()`) was introduced. This API allows it to request the information from which address a received message was transmitted from. The new API is necessary to be able to use a single SoAdSocketConnection for transmission and reception of messages without the drawback that received messages may be discarded because of not matching remote addresses.
Optional inclusion of a user configuration file (`SdUserConfigFile`).
Functionality of user callouts for configuration options as described in RfC 68746.
Additional configuration option to not configure unicast remote address of a SoAdSocketConnection which is configured for reception of multicast events.
Support of retry subscription mechanism for requested eventgroups as described in RfC 80174 and implemented in AUTOSAR4.4.0.
Support for grouping of services into ServiceGroups as described in AUTOSAR CP R19-11.
Support for sharing of SdServerTimers/SdClientTimers with a common random initial Offer/Find delay as described in AUTOSAR CP R19-11.
Reporting of runtime errors can be disabled which allows to avoid the otherwise mandatory reference to Det.h. Please note, that for this purpose development error reporting must also be disabled because it requires a reference to Det.h, too.
Instance IDs of Server Services and Client Services can be dynamically updated after initialization.
Multi-partition support for different SdInstances.
Support of functionality to subscribe to a multicast address pre-defined by a ClientService as described in AUTOSAR CP R21-11.

Table 2-3 Features provided beyond the referenced AUTOSAR Standard Release Version

2.1.3 Limitations

Table 2-4 shows general limitations that apply to the Sd.

Limitations
No general limitations known.

Table 2-4 General Limitations

2.1.4 Known Issues (low priority)

2.1.4.1 ESCAN00087299
Restricted functionality of compiler abstraction

The compiler abstraction for pointers does always use the identical 'ptrclass', independently from the memory location of the target. (It is not differentiated between variables stored in the pre-compile or post-build memory sections.)

Hence, the compiler abstraction cannot be used to specify and optimize pointers.

Workaround: Do not use special optimizations in compiler abstraction.

2.1.5 Dynamic update of Service Instance ID

The instance ID of a Server Service or Client Service is configured via BSWMD parameter SdServerServiceInstanceId or SdClientServiceInstanceId, respectively. After initialization of the SD module, the instance ID can be dynamically updated by calling the API function `Sd_ServerServiceSetInstanceId()` or `Sd_ClientServiceSetInstanceId()`, respectively.

If the Server Service or Client Service is configured to be automatically available or required, respectively, it is advised to call the API after `Sd_Init()`, but before `Sd_MainFunction()`. If the instance ID is updated after several main cycle functions have passed, Offer or Find messages with the configured instance ID might be already transmitted. In general, it is advised to update the instance ID before the service is activated and not during regular operation to avoid unexpected behavior of the SD module.

2.2 Initialization

The SD module is initialized via a `Sd_InitMemory()` call followed by a call of `Sd_PreInit()`. Afterwards, `Sd_Init()` must be called on each partition Sd is used on and initialization is finished by a call of `Sd_PostInit()`.
In all configuration Variants, the address of the global configuration data structure of the module must be passed to the `Sd_PreInit()` function.

Example

```
Sd_PreInit(Sd_Config_Ptr);
Sd_Init(Sd_Config_Ptr);
Sd_PostInit();
```

Figure 2-1 Initialization example for the Sd module

2.3 States

After initialization, the state of the SD instances is `SD_INSTANCE_DOWN`, because the Rx and Tx socket connections of the SD module are down. If the IP address is successfully configured, the state is indicated by a callback to the

`Sd_LocalIpAddrAssignmentChg()` function. This callback changes the state of the SD instances to `SD_INSTANCE_UP_AND_CONFIGURED` and initiates the client and server services to run.

2.4 Main Functions

The SD module has one or more main functions that need to be called periodically. This is normally done by the RTE. If no RTE is used, please call main function(s) manually. The number of main functions depends on whether multi-partition support is configured (see chapter 5.2.5.5). If it is disabled, only `Sd_MainFunction()` exists. Otherwise, one main function per partition is used with the naming pattern `Sd_MainFunction_<OsApplShortname>()`.

The main tasks of the main functions are:

- State handling for client and server services
- Generation and transmission of SD messages
- Processing of received SD messages
- Delayed execution of trigger transmit functionality

If multi-partition support is used, only the functionality of the respective partition is executed.

Note
The worst-case execution time regarding Rx Event processing time can be influenced by setting the maximum number of Rx queue entries which are processed during one Main Function call by changing config parameter `Sd/SdConfig/SdInstance/SdInQueueProcessLimit`

2.5 Error Handling

2.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [5], if the configuration parameter `SdDevErrorDetect` is set to TRUE in the SdGeneral container (see Figure 5-7 in chapter 5.2.6).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported SD ID is 171.

The reported service IDs are shown in the following table:

Service ID	Service
0x01u	`SD_SID_INIT`

Service ID	Service
0x02u	SD_SID_GET_VERSION_INFO
0x05u	SD_SID_LOCAL_IP_ADDR_ASSIGNMENT_CHG
0x06u	SD_SID_MAIN_FUNCTION
0x07u	SD_SID_SERVER_SERVICE_SET_STATE
0x08u	SD_SID_CLIENT_SERVICE_SET_STATE
0x09u	SD_SID_CONSUMED_EVENTGROUP_SET_STATE
0x42u	SD_SID_RX_INDICATION
0x43u	SD_SID_SOCONMODE_CHG
0x44u	SD_SID_SERVICE_GROUP_START
0x45u	SD_SID_SERVICE_GROUP_STOP
0xA2u	SD_SID_ADD_CLIENT_TO_LIST
0xA3u	SD_SID_SERIALIZE_PENDING_MESSAGES
0xA4u	SD_SID_COMMIT_SENDENTRY
0xA5u	SD_SID_GET_EMPTY_SENDENTRY
0xA6u	SD_SID_HANDLERETRYSUBSCRIPTION
0xA7u	SD_SID_GET_RESET_MEASURE_DATA
0xA8u	SD_SID_SERVER_SERVICE_SET_INSTANCE_ID
0xA9u	SD_SID_CLIENT_SERVICE_SET_INSTANCE_ID
0xAAu	SD_SID_PRE_INIT
0xABu	SD_SID_POST_INIT

Table 2-5 Service IDs

The development errors reported to DET are described in the following table:

Error Code		Description
0x00u	SD_E_NO_ERROR	No error occurred.
0x01u	SD_E_NOT_INITIALIZED	Module has not been initialized.
0x02u	SD_E_PARAM_POINTER	API service used with invalid pointer parameter.
0x03u	SD_E_INV_MODE	Invalid mode request.
0x04u	SD_E_INV_ID	Invalid ID.
0x05u	SD_E_INIT_FAILED	Unused error code.
0x06u	SD_E_COUNT_OF_RETRY_SUBSCRIPTION_EXCEEDED	Subscription retry was not successful.
0xA1u	SD_E_PARAM_CONFIG	API service Sd_Init() called with wrong parameter.

Error Code		Description
0xA2u	SD_E_INV_ARG	API service Sd_GetAndResetMeasurementData() called with invalid argument.
0xA3u	SD_E_INV_INSTANCE_ID	API service Sd_ServerServiceSetInstanceId() or Sd_ClientServiceSetInstanceId() called with invalid instance ID.
0xA4u	SD_E_NO_PREINIT	API service Sd_Init() or Sd_PostInit() called before pre-initialization.
0xA5u	SD_E_INV_APPLICATION_ID	Module API called with handle ID that does not match to OS application.
0xB1u	SD_E_ADDR_LIST_FULL	Internal address list is full. Check configuration parameter: SdMaxNrDestAddr.
0xB2u	SD_E_CLIENT_LIST_FULL	Internal client list is full. SdEventHandler is configured with insufficient amount of corresponding SoAdSocketConnection.
0xB3u	SD_E_TX_BUFFER_FULL	Size of SdEntry exceeds configured SdMesage Tx buffer size. TxEntry is lost. Check PDU size of SdTxPduRef or configuration parameter SdTxBufferSize.
0xB4u	SD_E_SEND_ENTRY_LIST_FULL	All reserved Tx SdEntries for this instance are occupied. TxEntry is lost.
0xB5u	SD_E_FREE_SEND_ENTRY_LIST_EMPTY	There are no free Tx SdEntries available. TxEntry is lost.

Table 2-6 Development Errors reported to DET

2.5.2 Runtime Error Reporting

By default, runtime errors are reported to the DET using the service Det_ReportRuntimeError() as specified in [5], if the configuration parameter SdRuntimeErrorReport is set to TRUE in the SdGeneral container (see Figure 5-7 in chapter 5.2.6).

The runtime errors reported to DET are described in the following table:

Error Code		Description
0x06u	SD_E_COUNT_OF_RETRY_SUB	Subscription retry failed after maximum number of attempts.

Table 2-7 Runtime Errors reported to DET

2.5.3 Production Code Error Reporting

By default, production code related errors are reported to the DEM using the service Dem_ReportErrorStatus() as specified in [6], if production error reporting is enabled (i.e., pre-compile parameter SD_DEM_EVENTS_CONFIGURED==STD_ON).

If another module is used for production code error reporting, the function prototype for reporting the error can be configured by the integrator but must have the same signature as the service `Dem_ReportErrorStatus()`.

The errors reported to DEM are described in the following table:

Error Code	Description
`SD_E_OUT_OF_RES`	SdEventHandler is not able to configure additional client
`SD_E_MALFORMED_MSG`	Malformed message received
`SD_E_SUBSCR_NACK_RECV`	The subscription for a SdConsumedEventGroup was rejected by a SubscribeEventgroupNack

Table 2-8 Errors reported to DEM

In order to clarify the individual DEM errors and to map them to the corresponding instance, the Handle ID of the SdInstance (`SD_E_OUT_OF_RES`, `SD_E_MALFORMED_MSG`) or the SdConsumedEventGroup (`SD_E_SUBSCR_NACK_RECV`) is added at the end of the define.

3 Integration

This chapter gives necessary information for the integration of the MICROSAR Classic SD into an application environment of an ECU.

3.1 Embedded Implementation

The delivery of the SD contains the files which are described in the chapters 3.1.1 and 3.1.2:

3.1.1 Static Files

File Name	Description
Sd.c	This is the source file of the SD module
Sd.h	API declaration of the module
Sd_Cbk.h	API declaration of SD callback functions
Sd_Priv.h	Component local macro and variable declaration
Sd_Types.h	Data type declarations

Table 3-1 Static files

3.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator Pro.

File Name	Description
Sd_Cfg.h	Pre-compile time parameter configuration.
Sd_Lcfg.c	Link-time parameter configuration.
Sd_Lcfg.h	Link-time parameter configuration declaration.
Sd_Lcfg_<OsAppl Shortname>.c	Partition-specific link-time parameter configuration. Only for multi-partition solution.
Sd_Lcfg_<OsAppl Shortname>.h	Partition-specific link-time parameter configuration declaration. Only for multi-partition solution.
Sd_PBcfg.c	Post-build parameter configuration.
Sd_PBcfg.h	Post-build parameter configuration declaration.
Sd_PBcfg_<OsAppl Shortname>.c	Partition-specific post-build parameter configuration. Only for multi-partition solution.
Sd_PBcfg_<OsAppl Shortname>.h	Partition-specific post-build parameter configuration declaration. Only for multi-partition solution.
Sd_MemMap.h	Memory section definitions (generated by the MemMap module).

Table 3-2 Generated files

3.2 Critical Sections

The SD module uses the Critical Section implementation of the SchM.

3.2.1 SD_EXCLUSIVE_AREA_0

SD_EXCLUSIVE_AREA_0
Purpose: Ensures consistency of partition- or instance-specific RAM variables. **Interfaces:** > SchM_Enter_Sd_SD_EXCLUSIVE_AREA_0 > SchM_Exit_Sd_SD_EXCLUSIVE_AREA_0 **Runtime:** Long: deep nested exclusive areas. **Dependency:** > Sd_Init() > Sd_MainFunctionPartition() > Sd_RxIndication() > Sd_GetAndResetMeasurementData() > Sd_ServerServiceSetState() > Sd_ClientServiceSetState() > Sd_ServiceGroupStart() > Sd_ServiceGroupStop() > Sd_ConsumedEventGroupSetState() **Recommendation:** It is recommended to use global interrupt locks for this critical section.

Table 3-3 SD_EXCLUSIVE_AREA_0

3.2.2 SD_EXCLUSIVE_AREA_MULTI_PARTITION

SD_EXCLUSIVE_AREA_ MULTI_PARTITION
Purpose: Ensures consistency of RAM variables, which are accessed from multiple partitions. **Interfaces:** > SchM_Enter_Sd_SD_EXCLUSIVE_AREA_MULTI_PARTITION

>	SchM_Exit_Sd_SD_EXCLUSIVE_AREA_MULTI_PARTITION

Runtime:
Short: Only a few statements with a constant maximum runtime.

Dependency:
> Sd_Init()

> Sd_RxIndication()

> Sd_GetAndResetMeasurementData()

Recommendation:
In case of a multi-partition use-case, it is required to implement this critical section as a spinlock. Otherwise, the same mechanism as used for SD_EXCLUSIVE_AREA_0 shall be used.

Table 3-4 SD_EXCLUSIVE_AREA_MULTI_PARTITION

Caution
Be aware that the SD calls the critical sections nested (with themselves and other critical sections) but exits the critical sections in the reverse order they were entered. Note that SD_EXCLUSIVE_AREA_MULTI_PARTITION is only nested with the other critical section. Please assert that this is supported by the implementation of the critical sections.

3.3 Preemption

Sd expects that its own main function does not interrupt the main functions of related modules (BswM, SoAd, TcpIp, …) and is not interrupted by main functions of related modules.

Caution
Consider main function expectations when using preemptive tasks.

3.4 Memory Sections

The Sd_MemMap.h file is generated by the MemMap Generator (`/ActiveEcuC/MemMap`). If adaptions should be done to the Memory Mapping of the Sd, the changes must be configured in the MemMap Generator.

3.4.1 Memory Sections for Multi-partition use-case

If the Sd is used in a multi-partition use-case, the RAM and ROM data will be generated into partition-specific memory sections.

The user must ensure that partitions cannot write into each other's memory (RO: Read only access). For partition-unspecific use-cases (e.g. measurement data), shared memory sections must be accessible (RW: read and write access) by all involved partitions. Table 3-5 gives an overview on the expected read and write accesses.

Memory Section	OsApplicationX	OsApplicationY
SD_OsApplicationX_<SectionType>	RW	RO
SD_OsApplicationY_<SectionType>	RO	RW
SD_<SectionType>	RW	RW

Table 3-5 Memory Mapping in multi-partition use-case

For the mapping of the partition-specific memory sections partition-specific software address methods (SwAddrMethods) are used. Refer to [9] for further details on SwAddrMethods. For the partition-specific SwAddrMethods the mapping is done automatically by the OS.

3.5 Multi-partition

If multi-partition support is used (see chapter 5.2.5.5) freedom from interference (FFI) regarding memory cannot be guaranteed.
Furthermore, all public interface functions of the SD module with handle ID as parameter must be called from context of the partition, where the configuration object of the handle is mapped to. Interface functions without handle ID can be called from any partition context. This includes:

- `Sd_InitMemory()`
- `Sd_Init()`
- `Sd_GetVersionInfo()`
- `Sd_GetAndResetMeasurementData()`

The functions `Sd_PreInit()` and `Sd_PostInit()` must be called from context of the main partition.

4 API Description

For an interfaces overview please see [1].

4.1 Type Definitions

The types defined by the SD are described in this chapter.

Type Name	C-Type	Description	Value Range
Sd_ServerServiceSetStateType	Enum	Server states that are reported to the SD using the expected API Sd_ServerServiceSetState	SD_SERVER_SERVICE_DOWN Server service is not requested
			SD_SERVER_SERVICE_AVAILABLE Server service is requested
Sd_ClientServiceSetStateType	Enum	Client states that are reported to the BswM using the expected API Sd_ClientServiceSetState	SD_CLIENT_SERVICE_RELEASED Client service is not requested
			SD_CLIENT_SERVICE_REQUESTED Client service is requested
Sd_ConsumedEventGroupSetStateType	Enum	Subscription policy by consumed EventGroup for the Client Service	SD_CONSUMED_EVENTGROUP_RELEASED ConsumedEventGroup is not requested
			SD_CONSUMED_EVENTGROUP_REQUESTED ConsumedEventGroup is requested
Sd_ClientServiceCurrentStateType	Enum	Modes to indicate the current mode request of a Client Service	SD_CLIENT_SERVICE_DOWN Client Service is not running
			SD_CLIENT_SERVICE_AVAILABLE Client Service is running now
Sd_ConsumedEventGroupCurrentStateType	Enum	Subscription policy by consumed EventGroup for the Client Service	SD_CONSUMED_EVENTGROUP_DOWN The EventGroup cannot be reached
			SD_CONSUMED_EVENTGROUP_AVAILABLE Successfully subscribed for the EventGroup
Sd_EventHandlerCurrentStateType	Enum	Subscription policy by EventHandler for the Server Service	SD_EVENT_HANDLER_RELEASED The EventHandler is not requested
			SD_EVENT_HANDLER_REQUESTED The EventHandler is requested

Sd_ConfigOptionStringType	const uint8 *	Type for a zero-terminated string of configuration options.	

Table 4-1 Type definitions

4.2 Services provided by SD

4.2.1 Sd_InitMemory

Prototype	
`void Sd_InitMemory (void)`	
Parameter	
`void`	none
Return Code	
void	none
Functional Description	
This function is used to initialize the global variables of the Sd at startup if not done by startup code.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
	**Caution** This function shall be called before Sd_Init.
Pre-Conditions	
none	
Call Context	
This function shall be called for initialization.	

Table 4-2 Sd_InitMemory

4.2.2 Sd_PreInit

Prototype	
`void Sd_PreInit ( const Sd_ConfigType* ConfigPtr)`	
Parameter	
`ConfigPtr`	Configuration structure for initializing the module.
Return Code	
void	none
Functional Description	
This function initializes the shared memory of Sd module and sets the module to the pre-initialized state.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
	**Caution** This function has to be called before usage of the module.
Pre-Conditions	
`Sd_InitMemory()` has been called unless global variables are initialized in start-up code. Interrupts are disabled.	
Call Context	
This function shall be called for initialization.	

Table 4-3 Sd_PreInit

4.2.3 Sd_Init

Prototype	
`void Sd_Init ( const Sd_ConfigType* ConfigPtr)`	
Parameter	
`ConfigPtr`	Configuration structure for initializing the module. This parameter is not used since it is already provided in context of Sd_PreInit().
Return Code	
void	none
Functional Description	
This function initializes all variables of one partition and sets the initalization state of the partition.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
	**Caution** This function has to be called before usage of the module.
Pre-Conditions	
`Sd_PreInit()` has been called. Interrupts are disabled.	
Call Context	
This function shall be called for initialization. In multi-partition use-case: This function has to be called once in each partition context.	

Table 4-4 Sd_Init

4.2.4 Sd_PostInit

Prototype	
`void` **`Sd_PostInit`** `(void)`	
Parameter	
`void`	none
Return Code	
void	none
Functional Description	
This function checks the initialization state of all partitions and sets the global initialized state.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
	**Caution** This function has to be called before usage of the module.
Pre-Conditions	
`Sd_Init()` has been called on each partition.	
Call Context	
This function shall be called for initialization. This function must be called only once and it must be called on the main partition in case of multi-partition.	

Table 4-5 Sd_PostInit

4.2.5 Sd_ServerServiceSetState

Prototype	
`Std_ReturnType Sd_ServerServiceSetState ( uint16 ServerServiceHandleId, Sd_ServerServiceSetStateType ServerServiceState)`	
Parameter	
`ServerServiceHandleId`	ID to identify the Server Service Instance.
`ServerServiceState`	The state the Server Service Instance shall be set to.
Return Code	
Std_ReturnType	E_OK: State accepted E_NOT_OK: State not accepted
Functional Description	
This API function is used by the BswM to set the Server Service Instance state.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-6 Sd_ServerServiceSetState

4.2.6 Sd_ClientServiceSetState

Prototype	
`Std_ReturnType Sd_ClientServiceSetState ( uint16 ClientServiceHandleID, Sd_ClientServiceSetStateType ClientServiceState)`	
Parameter	
`ClientServiceHandleID`	ID to identify the Client Service Instance.
`ClientServiceState`	The state the Client Service Instance shall be set to.
Return Code	
Std_ReturnType	E_OK: State accepted E_NOT_OK: State not accepted
Functional Description	
This API function is used by the BswM to set the Client Service Instance state.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-7 Sd_ClientServiceSetState

4.2.7 Sd_ServerServiceSetInstanceId

Prototype	

`Std_ReturnType` **`Sd_ServerServiceSetInstanceId`** `(`
`  uint16 ServerServiceHandleId,`
`  uint16 ServerServiceInstanceId)`	

Parameter	
`ServerServiceHandleId`	ID to identify the Server Service Instance.
`ServerServiceInstanceId`	New server service instance ID
Return Code	
Std_ReturnType	E_OK: Server service instance ID was changed E_NOT_OK: Server service instance ID was not changed
Functional Description	
This API function is used by the BswM to assign a new Server Service Instance ID.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-8 Sd_ServerServiceSetInstanceId

4.2.8 Sd_ClientServiceSetInstanceId

Prototype	

`Std_ReturnType` **`Sd_ClientServiceSetInstanceId`** `(`
`  uint16 ClientServiceHandleID,`
`  uint16 ClientServiceInstanceId)`	

Parameter	
`ClientServiceHandleID`	ID to identify the Client Service Instance.
`ClientServiceInstanceId`	New client service instance ID
Return Code	
Std_ReturnType	E_OK: Client service instance ID was changed E_NOT_OK: Client service instance ID was not changed
Functional Description	
This API function is used by the BswM to assign a new Client Service Instance ID.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-9 Sd_ClientServiceSetInstanceId

4.2.9 Sd_ConsumedEventGroupSetState

Prototype	

`Std_ReturnType` **`Sd_ConsumedEventGroupSetState`** `(`
`uint16 ConsumedEventGroupHandleId,`
`Sd_ConsumedEventGroupSetStateType ConsumedEventGroupState)`	

Parameter	
`ConsumedEventGroupHandleId`	ID to identify the ConsumedEventGroupHandleId
`ConsumedEventGroupState`	The state the EventGroup shall be set to
Return Code	
Std_ReturnType	E_OK: State accepted E_NOT_OK: State not accepted
Functional Description	
This API function is used by the BswM to set the requested state of the EventGroupStatus.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-10 Sd_ConsumedEventGroupSetState

4.2.10 Sd_ServiceGroupStart

Prototype	
`void Sd_ServiceGroupStart ( uint16 ServiceGroupHandleId)`	
Parameter	
`ServiceGroupHandleId`	ID of the ServiceGroup which shall be started.
Return code	
-	-
Functional Description	
Sets all members of the specified SdServiceGroup to REQUESTED / AVAILABLE.	
Particularities and Limitations	
> Module has to be initialized. > API uses an internal request counter, see Sd_ServiceGroupStop().	
Call context	
Task level.	

Table 4-11 Sd_ServiceGroupStart

4.2.11 Sd_ServiceGroupStop

Prototype	
`void Sd_ServiceGroupStop ( uint16 ServiceGroupHandleId)`	
Parameter	
`ServiceGroupHandleId`	ID of the ServiceGroup which shall be started.
Return code	
-	-
Functional Description	
Sets all members of the specified SdServiceGroup to RELEASED / DOWN.	
Particularities and Limitations	
> Module has to be initialized. > When using the ServiceGroup API, an internal request counter is increased (Start) / decreased (Stop) for each service which is a member of the associated group. A service is only set to RELEASED/DOWN if the request counter reaches zero. It is expected from the caller to keep track of the state of each ServiceGroup and call the API symmetrically.	
Call context	
Task level.	

Table 4-12 Sd_ServiceGroupStop

4.2.12 Sd_MainFunction

Prototype	
`void` **`Sd_MainFunction`** `(void)`	
Parameter	
`void`	none
Return Code	
void	none
Functional Description	
Periodically called MainFunction that handles the Sd state for each service instance.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs If multi-partition support is used (see chapter 5.2.5.5), the MainFunction exists per partition with the naming pattern: `Sd_MainFunction_<OsApplShortname>()`.	
Pre-Conditions	
Sd_InitMemory(), Sd_PreInit(), Sd_Init() and Sd_PostInit() must have been called first.	
Call Context	
Task level	

Table 4-13 Sd_MainFunction

4.3 Services used by SD

In the following table services provided by other components, which are used by the SD are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DET	Det_ReportError
DEM	Dem_ReportErrorStatus
BswM	BswM_Sd_ClientServiceCurrentState
BswM	BswM_Sd_ConsumedEventGroupCurrentState
BswM	BswM_Sd_EventHandlerCurrentState
Os	GetApplicationId
SoAd	SoAd_CloseSoCon
SoAd	SoAd_DisableRouting
SoAd	SoAd_DisableSpecificRouting
SoAd	SoAd_EnableRouting
SoAd	SoAd_EnableSpecificRouting
SoAd	SoAd_ForceReleaseRemoteAddr
SoAd	SoAd_GetLocalAddr
SoAd	SoAd_GetPhysAddr
SoAd	SoAd_GetRcvRemoteAddr
SoAd	SoAd_GetRemoteAddr
SoAd	SoAd_GetSoConId
SoAd	SoAd_IfRoutingGroupTransmit
SoAd	SoAd_IfSpecificRoutingGroupTransmit
SoAd	SoAd_OpenSoCon
SoAd	SoAd_ReleaseIpAddrAssignment
SoAd	SoAd_RequestIpAddrAssignment
SoAd	SoAd_SetRemoteAddr
SoAd	SoAd_SetUniqueRemoteAddr

Table 4-14 Services used by the SD

4.4 Callback Functions

This chapter describes the callback functions that are implemented by the SD and can be invoked by other modules. The prototypes of the callback functions are provided in the header file `Sd_Cbk.h`.

4.4.1 Sd_RxIndication

Prototype	
`void Sd_RxIndication ( PduIdType RxPduId, const PduInfoType *PduInfoPtr)`	
Parameter	
`RxPduId`	ID of the received I-PDU.
`PduInfoPtr`	Contains the length (SduLength) of the received I-PDU and a pointer to a buffer (SduDataPtr) containing the I-PDU.
Return Code	
void	none
Functional Description	
Indication of a received I-PDU from a lower layer communication interface module.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
The function is called by the SoAd.	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Interrupt level	

Table 4-15 Sd_RxIndication

4.4.2 Sd_LocalIpAddrAssignmentChg

Prototype	
`void Sd_LocalIpAddrAssignmentChg ( SoAd_SoConIdType SoConId, SoAd_IpAddrStateType State)`	
Parameter	
`SoConId`	Socket connection index specifying the socket connection where the IP address assignment has changed.
`State`	State of IP address assignment.
Return Code	
void	none
Functional Description	
This function gets called by the SoAd if an IP address assignment related to a socket connection changes (i.e. new address assigned or assigned address becomes invalid).	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-16 Sd_LocalIpAddrAssignmentChg

4.4.3 Sd_SoConModeChg

Prototype	
`void Sd_SoConModeChg ( SoAd_SoConIdType SoConId, SoAd_SoConModeType Mode)`	
Parameter	
`SoConId`	Socket connection index specifying the socket connection with the mode change.
`Mode`	New socket connection mode.
Return Code	
void	none
Functional Description	
This callback is called by the SoAd module if the socket connection state has changed.	
Particularities and Limitations	
The function is called by the SoAd.	
Pre-Conditions	
Module has to be initialized.	
Call Context	
Task level	

Table 4-17 Sd_SoConModeChg

4.5 Callout Functions

This chapter describes the callout functions which can be invoked by the SD. The prototypes of the callout functions are provided in the header file Sd_Lcfg.h.

4.5.1 <SdCapabilityRecordMatchCalloutName>

Prototype

`boolean <SdCapabilityRecordMatchCalloutName>(`
`  PduIdType pduID,`
`  uint8 type,`
`  uint16 serviceID,`
`  uint16 instanceID,`
`  uint8 majorVersion,`
`  uint32 minorVersion,`
`  const Sd_ConfigOptionStringType * receivedConfigOptionPtrArray,`
`  const Sd_ConfigOptionStringType * configuredConfigOptionPtrArray)`

Parameter	
`pduID`	ID of the received I-PDU (used to distinguish between different SD instances)
`type`	Content of the Type field of the received entry.
`serviceID`	Content of the Service ID field of the received entry.
`instanceID`	Content of the Instance ID field of the received entry.
`majorVersion`	Content of the Major Version field of the received entry.
`minorVersion`	Content of the Minor Version field of the received entry.
`receivedConfigOptionPtrArray`	`NULL_PTR` terminated array of pointers to zero-terminated configuration strings received in the incoming entry, i.e., received SD message.
`configuredConfigOptionPtrArray`	`NULL_PTR` terminated array of pointers to zero-terminated configuration strings configured in the local SD configuration.

Return Code	
boolean	True: Received configuration options shall be accepted. False: The received configuration options are invalid.

Functional Description
This callout is invoked to determine whether the configuration options contained in a received SD message match the ones configured in the local SD configuration (i.e., SdServerCapabilityRecord or SdClientCapabilityRecord).

Particularities and Limitations
Pre-Conditions
> Availability: This function(s) are only available if configured in the `Sd/SdConfig/SdCapabilityRecordMatchCallout` container.
Call Context
This function is called in interrupt context.

Table 4-18 <SdCapabilityRecordMatchCalloutName>

4.6 Configurable Interfaces

4.6.1 Sd_GetVersionInfo

Prototype	
`void Sd_GetVersionInfo ( Std_VersionInfoType* VersionInfoPtr)`	
Parameter	
`VersionInfoPtr`	Pointer to a memory location where the Sd version information shall be stored.
Return Code	
void	none
Functional Description	
Returns version information, vendor ID and AUTOSAR module ID of the component. The versions are BCD-coded.	
Particularities and Limitations	
Service ID: see Table 2-5 Service IDs	
Pre-Conditions	
> Availability: This function is only available if SdVersionInfoApi is enabled.	
Call Context	
This function can be called in any context.	

Table 4-19 Sd_GetVersionInfo

4.6.2 Sd_GetAndResetMeasurementData

Prototype	
`Std_ReturnType` **`Sd_GetAndResetMeasurementData`** `( Sd_MeasurementIdxType MeasurementIdx, boolean MeasurementResetNeeded, uint32* MeasurementDataPtr)`	
Parameter	
`MeasurementIdx`	Index of the measurement data value to retrieve and/or reset. SD_MEAS_ALL can only be used to reset all measurement data values with `MeasurementResetNeeded` = TRUE. No counter value will be read back in this case and `MeasurementDataPtr` should be NULL_PTR.
`MeasurementResetNeeded`	Indicates if the counter value shall be reset.
`MeasurementDataPtr`	Buffer where the value of the counter is to be copied into. May be NULL_PTR for only resetting measurement values.
Return code	
Std_ReturnType	E_OK: Operations successful. E_NOT_OK: Operations failed.
Functional Description	
Gets the value of (and/or resets) a measurement data counter.	
Particularities and Limitations	
> Availability: This function is only available if SdGetAndResetMeasurementDataApi is enabled.	
Call context	
This function can be called in any context.	

Table 4-20 Sd_GetAndResetMeasurementData

5 Configuration

In the SD the attributes can be configured with the tool DaVinci Configurator Pro.

5.1 Configuration Variants

The SD supports the configuration variants

- VARIANT-PRE-COMPILE
- VARIANT-POST-BUILD-LOADABLE
- VARIANT-POST-BUILD- SELECTABLE

The configuration classes of the SD parameters depend on the supported configuration variants. For their definitions please see the Sd_bswmd.arxml file.

5.2 Configuration with DaVinci Configurator Pro

The Service Discovery module is configured with the help of the configuration tool Configurator Pro. The configuration tool GENy is not supported by this MICROSAR Classic version.

In the following sub-chapters, some configuration hints are given to understand how to configure the SD correctly.

5.2.1 SD Communication Path

Each instance of the module (`SdInstance`) requires a proper configuration of its communication paths. The corresponding containers are the `SdInstanceTxPdu` for data transmission as well as `SdInstanceUnicastRxPdu` and `SdInstanceMulticastRxPdu` for data reception.

Within these containers, a `Pdu` has to be referenced.

Note
The `SdInstanceTxPdu`, `SdInstanceUnicastRxPdu` and `SdInstanceMulticastRxPdu` are used by the SD to communicate with its lower layer module SoAd. For these PDUs no configuration in the PduR and Com is required.

5.2.1.1 Tx Data Path

In order to configure the entire data path for data transmission, the referenced `Pdu` needs additional configuration. This includes the presence of a `SoAdPduRoute`, `SoAdSocketConnection` and a lower layer socket configuration.

According to the SD specification, the first bytes of each SD message are set by the SoAd by configuring the PDU header ID to 0xFFFF8100. (`SoAdPduHeaderEnable`, `SoAdTxPduHeaderId`)

For details about the SoAd configuration please see [3].

5.2.1.2 Rx Data Path

The Rx data path of the SD has to differentiate between messages received by unicast or multicast. This differentiation is implemented by two independent data paths. Each of them is represented by a `SoAdSocketRoute`, `SoAdSocketConnection` and a lower layer socket configuration.

As well as in the Tx data path, the SD uses the PDU header functionality also during reception of messages. (`SoAdPduHeaderEnable`, `SoAdTxPduHeaderId`)

The differentiation between unicast and multicast messages is necessary to implement a sophisticated response behavior. The response messages (of different ECUs) which have their origin in a multicast message are transmitted with an additional delay. This delay extends the distribution of messages on the transmission medium and reduces overload situations.

The amount of delay, which is added to the message transmission, is determined randomly within the range of the appropriate parameters.

- `SdServerTimerRequestResponseMinDelay` and `SdServerTimerRequestResponseMaxDelay`
- `SdClientTimerRequestResponseMinDelay` and `SdClientTimerRequestResponseMaxDelay`

The random number function has to be defined by the customer. See chapter 5.2.6.

5.2.2 General Information

The SD module provides two main functionalities.

- To publish (server) and find/subscribe (client) available services.
- The configuration of the SOAD-internal routing paths for SOMEIP messages.

In order to perform the configuration of the SOAD-internal routing paths, the SD module has the possibility to enable and disable routing paths as well as to configure the remote IP addresses of the communication partner. Additionally, a client service is able to configure the local multicast address of an eventgroup.

Note
The SD module is only able to change the configuration of a local or remote address if the corresponding parameters are not preconfigured. Hence, if the IP address is not set and the Ports are not configured (remote) or set to 0 (local).

Note
All SOMEIP communication paths have to be configured completely. Only the remote and local IP addresses can be adapted dynamically during runtime.

The enabling and disabling of the communication paths is handled via SoAdRoutingGroups, which are referenced within the SD configuration. Each independent routing group represents thereby either:

> All methods provided/used by this service, or

> A single eventgroup in combination with a transport paradigm (TCP, UDP-Unicast/Multicast)

5.2.3 Server Service

A server service is represented by the SdServerService container (Figure 5-1).

If this specific server instance should offer the functionality of SOMEIP events and/or methods within the network, additional SdEventHandler containers and/or a single SdProvidedMethods container must be configured.

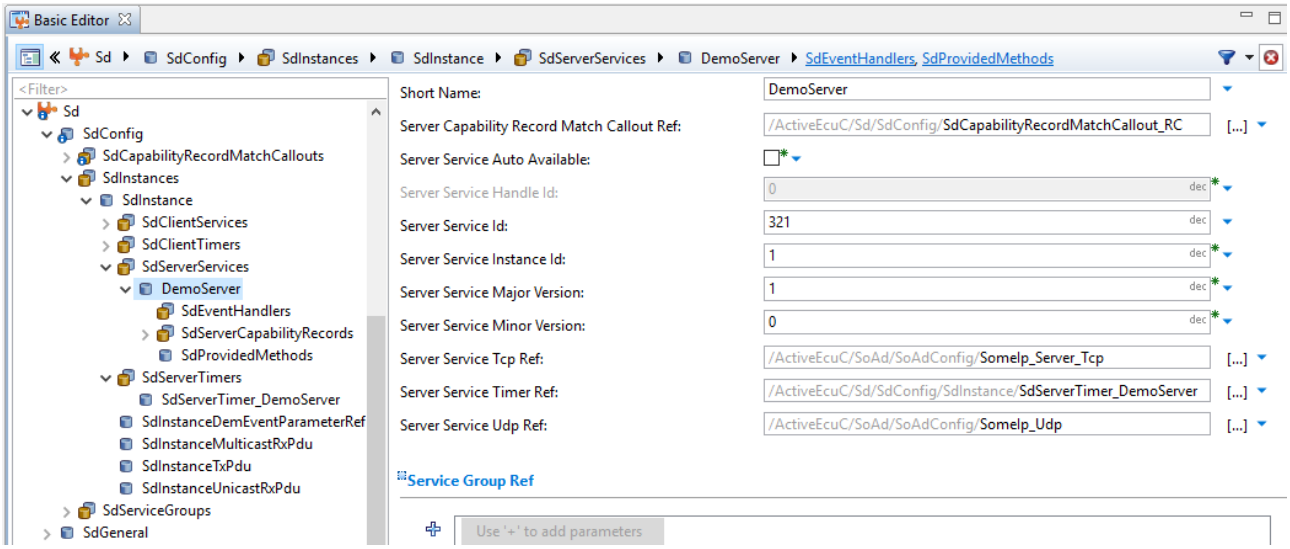

Figure 5-1 SdServerService container

5.2.3.1 Availability of a Server Service

A server service is not available after startup by default and can be made available with the Sd_ServerServiceSetState API or Sd_ServiceGroupStart API if it references a service group. If SdServerServiceAutoAvailable is enabled, the server service will be made available automatically after startup and must not be member of any service group.

5.2.3.2 Event Handler

The SdEventHandler provides the functionality of eventgroups. Each eventgroup is characterized by a set of SOMEIP events. The mapping between event and event group is realized indirectly within the SoAdSocketRoute and SoAdPduRoute containers.

Each event which is transmitted by the SOMEIP has an individual SoAdPduRoute with the corresponding PDU Header ID and one or multiple SoAdPduRouteDests. Thereby the SOAD implements a fan-out from a single TxPdu to multiple destinations such as socket connections or socket connection groups. (see Figure 5-2 SoAdPduRoute container).

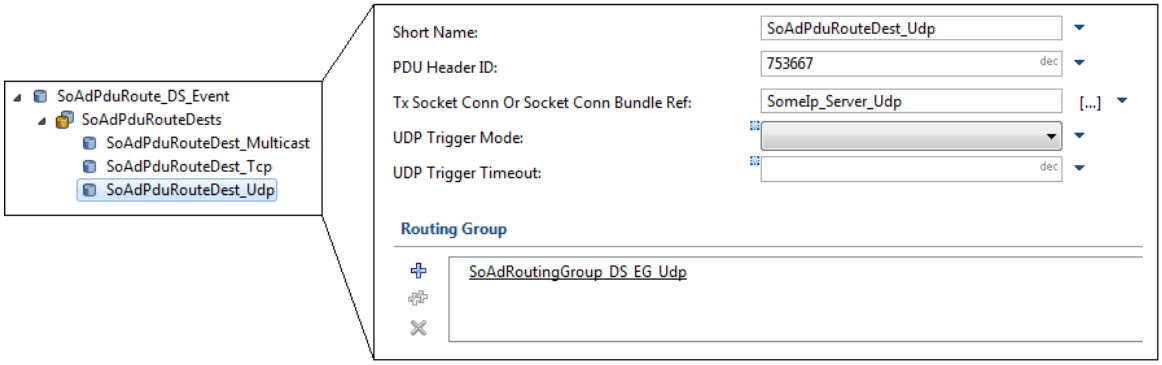

Figure 5-2 SoAdPduRoute container

The routing groups are therefore referenced by the `SdEventActivationRef` which is part of the `SdEventHandlerUdp`, `SdEventHandlerMulticast` and `SdEventHandlerTcp` containers shown in Figure 5-3.

The `SdEventTriggeringRef` provides the trigger transmit functionality. Thereby, the SD module triggers an additional transmission of all corresponding TxPdus if a new client subscribes to the eventgroup. This functionality is used to provide new clients with the actual values of the events without waiting for the next transmission of the event by the application. The trigger transmit functionality is only available for unicast communication paths and not for multicast.

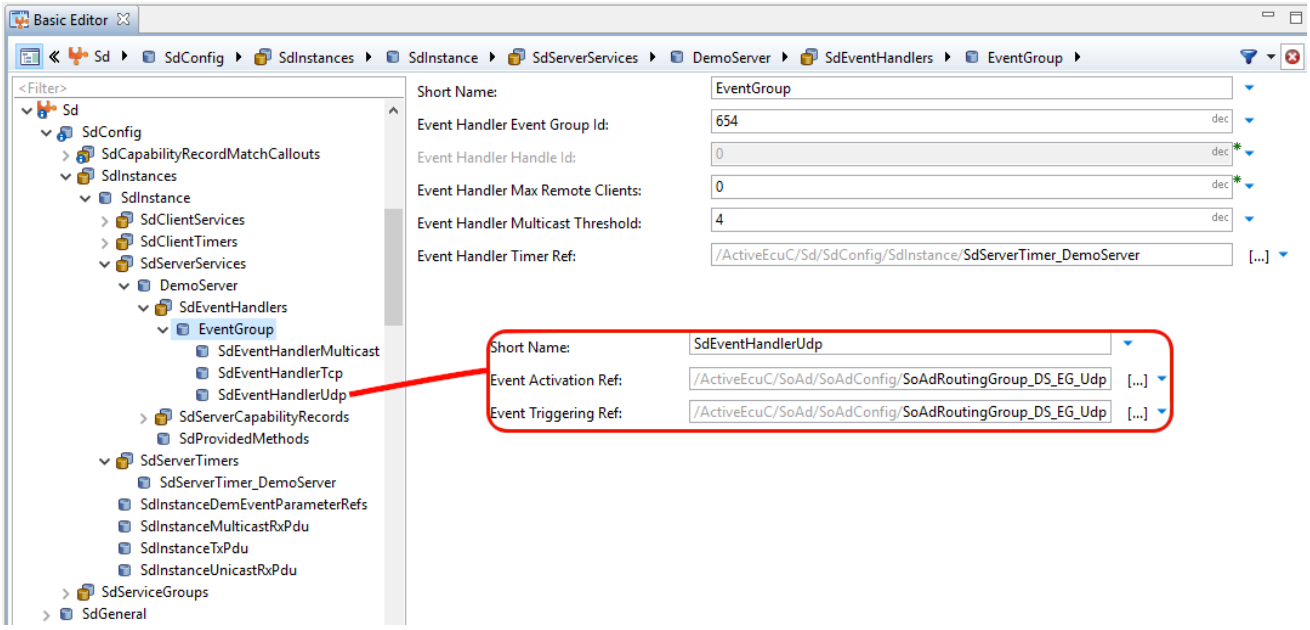

Figure 5-3 SdEventHandler container

The `SdEventHandlerMulticastThreshold` parameter specifies how many clients will be provided with the events by UDP unicast until the SD module changes the communication to multicast.

Note
The maximum number of remote clients for an event handler is determined automatically according to the number of referenced unicast SocketConnections or, if no unicast communication is configured, according to the parameter `SdMaxNrDestAddr` which is configurable per `SdInstance`.

However, the calculated value can be overwritten by setting `SdEventHandlerMaxRemoteClients` to != 0 to save RAM or to allow for more clients for certain event handlers.

5.2.3.3 Provided Methods

The `SdProvidedMethods` container contains an activation reference to the corresponding routing group. The socket connections used for reception and transmission of methods are referenced within the `SdServerService` container.

5.2.4 Client Service

A client service is represented by the `SdClientService` container (Figure 5-4).

If this specific client instance should use the functionality of SOMEIP events and/or methods within the network, additional `SdConsumedEventGroup` containers and/or a single `SdConsumedMethods` container must be configured.

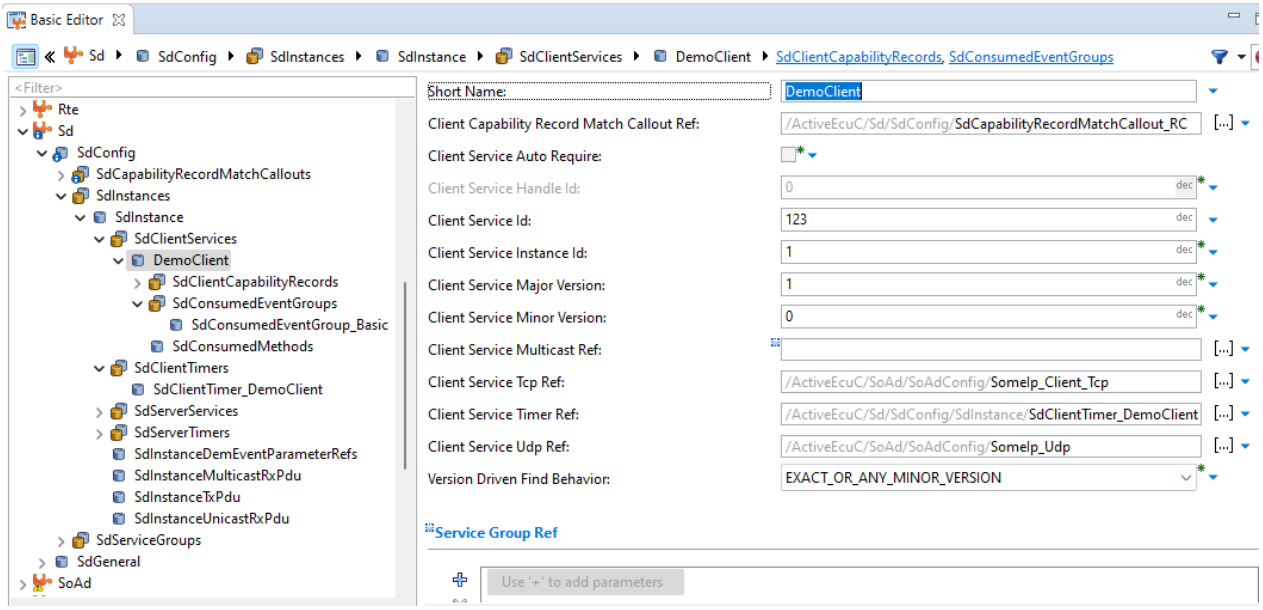

Figure 5-4 SdClientService container

5.2.4.1 Auto Require

By default, a client service must be explicitly set to required by calling the Sd_ClientServiceSetState or Sd_ServiceGroupStart API respectively, if the client service is member of a service group. If `SdClientServiceAutoRequire` is enabled however, a

client service must not be part of any service group and is set to required automatically after startup.

5.2.4.2 Version Driven Find Behavior

`SdVersionDrivenFindBehavior` specifies which minor versions are accepted for a client service when OfferService entries are received.

- `EXACT_OR_ANY_MINOR_VERSION`: If `SdClientServiceMinorVersion` is set to any value except wildcard (0xFFFFFFFF), the minor version of the offered server service must match to this value. If it is set to wildcard, any offered minor version will be accepted.
- `MINIMUM_MINOR_VERSION`: Configure `SdClientServiceMinorVersion` to the minimum minor version which shall be accepted. Offered server services must have a minor version equal or greater than the configured value.

The minor version in sent FindService entries is set to the configured `SdClientServiceMinorVersion` if `SdVersionDrivenFindBehavior` is set to `EXACT_OR_ANY_MINOR_VERSION` or to wildcard if this parameter is set to `MINIMUM_MINOR_VERSION`.

5.2.4.3 ClientService Multicast Ref

`SdClientServiceMulticastRef` has been introduced by AUTOSAR CP R21-11 and, if set, enables ConsumedEventGroups of the Service to use a multicast endpoint which is pre-defined by the ClientService. Unicast UDP and ClientService Multicast is mutually exclusive. However, ClientMulticast can be used in conjunction with EventHandler Multicast.

Caution
Using `SdClientServiceMulticastRef` affects ConsumedMethods and Initial Events; these features might not work as intended. This has been accepted during the specification of the feature.

5.2.4.4 Consumed Event Group

The `SdConsumedEventGroup` implements the functionality of eventgroups. Each eventgroup is characterized by a set of SOMEIP events. The mapping between event and event group is realized indirectly within the `SoAdSocketRoute` and `SoAdPduRoute` containers.

Each event which is received by the SOMEIP has an individual `SoAdSocketRoute` with the corresponding PDU Header ID per possible data reception path. The referenced routing groups correspond to the consumed eventgroups which contain the events. As shown in Figure 5-5, each `SdConsumedEventGroup` contains references to the routing groups in order to de/activate the reception of the corresponding events.

A successful subscription and acknowledge has to contain all information which are required by the consumed event group, in order to configure the available communication paths (TCP, UDP unicast and multicast).

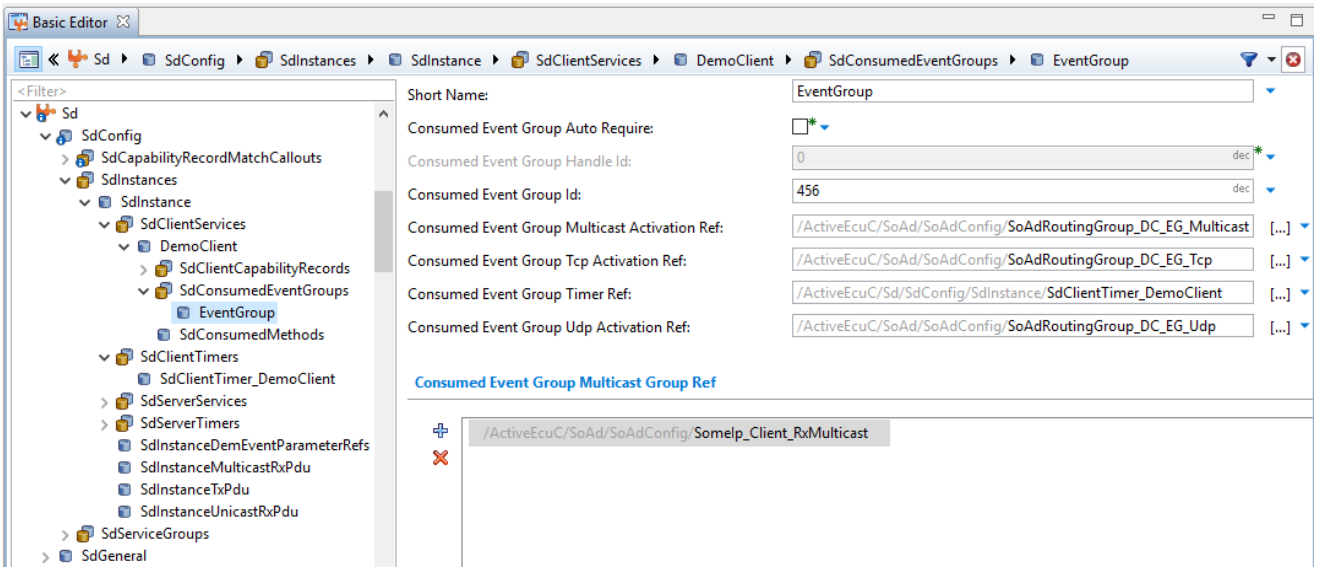

Figure 5-5 SdConsumedEventGroup container

The SD module can configure the multicast paths dynamically during runtime. This possibility comprises the configuration of the local multicast IP address and port and optionally the configuration of the remote IP address and port. The corresponding local IP address must be configured with the address type “Multicast”.

The optional configuration of the socket remote address can be enabled and disabled by the `SdSetRemAddrOfClientRxMulticastSoCon` parameter, which is located in the `SdGeneral` container. The default value (ENABLED) leads to a behavior which is conform to the AUTOSAR specification [1]. In this configuration, the remote IP address and port of the server are configured at the corresponding `SoAdSocketConnection`, which allows only a reception of messages from this particular remote server. Hence, the configuration has to contain an individual `SoAdSocketConnection` for each remote server which shall be used in parallel. If the `SdSetRemAddrOfClientRxMulticastSoCon` parameter is disabled, the remote IP address and port of the multicast connection are not set by the SD module. If the remote IP address and/or the port are configured to WILDCARD, this connection can be used to receive messages from multiple remote servers using multicast.

Starting with version 8.00.00, the module supports also the retry subscription mechanism for requested eventgroups which is described in AUTOSAR RfC80174 and implemented with AUTOSAR4.4.0. This mechanism was introduced in order to increase the robustness of the service discovery communication with respect to packet loss. If a client is configured for subscription retries via the configuration parameters `SdSubscribeEventgroupRetryMax` and `SdSubscribeEventgroupRetryDelay`, the client will perform subscription retires if it does not receive any response from the server.

Caution
It is not possible to configure multiple local addresses with the identical multicast IP address. Neither as preconfigured address nor as dynamically assigned address during runtime. Therefore, the user has to ensure that each eventgroup which is configured for multicast contains valid communication paths to all required and during runtime requested IP addresses. Therefore, multiple `SdConsumedEventGroupMulticastGroupRef` references can be configured.

5.2.4.5 Consumed Method

The `SdConsumedMethods` container contains an activation reference to the corresponding routing group. The socket connections used for reception and transmission of methods are referenced within the `SdClientService` container.

5.2.5 SdInstance Container

A local Service Discovery instance is represented by a `SdInstance` container (Figure 5-6). Each instance is an entity which is implementing the Sd protocol on a local interface and is independent from other instances. This chapter describes noteworthy parameters of the `SdInstance` container.

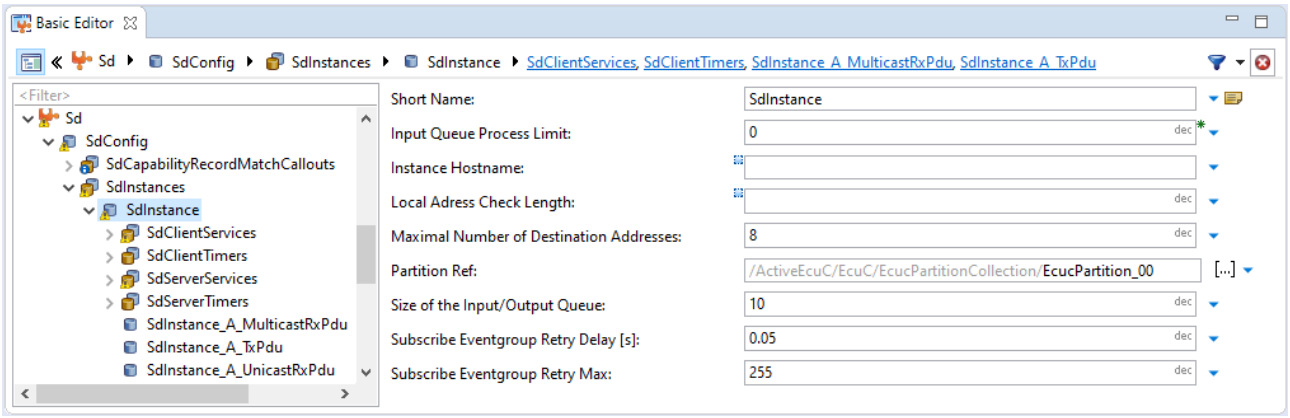

Figure 5-6 SdInstance Container

5.2.5.1 Rx/Tx Queue Configuration

`SdInOutQueueSize` determines how many SdEntries can be queued for Rx/Tx processing. If `SdInQueueProcessLimit` is set to a value greater than zero, this value specifies the maximum number of RxQueue entries which will be processed in one MainFunction.

Note
While the RxQueue is only used to store (Stop)Subscribe SdEntries, the TxQueue is used to store all SdEntries which shall be sent by the SdInstance.

5.2.5.2 Subscription Retry Mechanism

If the retry mechanism is enabled globally with `SdSubscribeEventgroupRetryEnable` in the `SdGeneral` container, parameter `SdSubscribeEventgroupRetryDelay` controls the interval in which SubscribeEntries shall be sent to retry the subscription and `SdSubscribeEventgroupRetryMax` specifies the maximum number of retry attempts for this individual instance. If `SdSubscribeEventgroupRetryMax` is set to 255, this means that no maximum number of retries shall be used for this SdInstance.

5.2.5.3 SdMaxNrDestAddr

`SdMaxNrDestAddr` specifies the maximum number of remote SD instances the local SD instance can communicate with. If the maximum is reached, SdMessages from ‘new’ instances which are not in the list will be ignored.

5.2.5.4 SdInstanceLocalAdressCheckLength

Endpoints that are received in SdMessages shall be checked for topological correctness. There are two options how to determine if an Endpoint shall be accepted or not:

> If parameter SdInstanceLocalAdressCheckLength does not exist in the configuration, the configured netmask of the local address is used to check all received Endpoints; SD communication will be restricted to the configured subnet. This is the default behavior.

> If SdInstanceLocalAdressCheckLength does exist, the value of this parameter is interpreted as CIDR network prefix length to check the correctness of the received Endpoints. This might be useful for use cases for which SD Communication shall also be possible with other subnets which are connect to the local SdInstance via a gateway or router.

5.2.5.5 PartitionRef / multi-partition support

SdInstances can be mapped to different partitions using the reference parameter SdPartitionRef. If two or more different partitions are referenced by all SdInstances, the multi-partition use case applies. The transmission buffer (see chapter 5.2.6) is then duplicated for each partition and accessed separately by the respective main function (see chapter 2.4).

For the multi-partition use case, it is important to ensure that the PDU, which is referenced by SdInstanceMulticastRxPdu, SdInstanceUnicastRxPdu and SdInstanceTxPdu, references the same partition as the SdInstance. Furthermore, the SdClientServices and SdServerServices of an SdInstance can only use a SoAdSocketConnectionGroup that is referenced by a SoAdInstance, which in turn references the same partition as the SdInstance.

5.2.6 SdGeneral Container

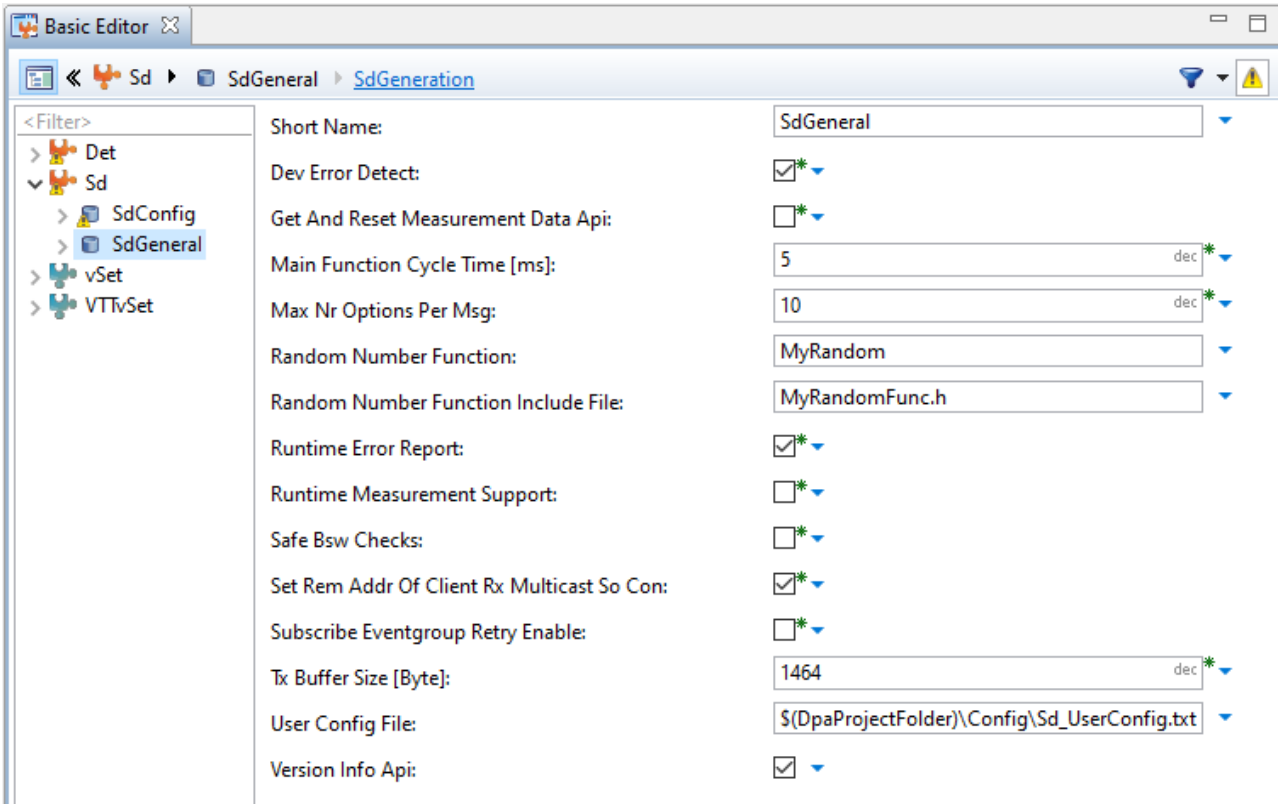

Figure 5-7 Basic Editor showing the SdGeneral container

The `SdGeneral` container shown in Figure 5-7 contains configuration parameters which are relevant for the entire SD module. A detailed description of each configuration parameter can be found in the description view of the parameter properties.

5.2.6.1 Random Number Function

The `SdRandomNumberFunctionIncludeFile` and `SdRandomNumberFunction` have to be provided by the customer. The random number function is used by the SD module in order to delay outgoing response messages which have their origin in messages transmitted by multicast.

5.2.6.2 Measurement API

If parameter `SdGetAndResetMeasurementDataApi` is enabled, diagnostic measurement data can be retrieved by API Sd_GetAndResetMeasurementData.

5.2.6.3 Runtime Measurement Support

Parameter `SdRuntimeMeasurementSupport` can only be set if the Rtm module is part of the configuration. If it is set to enabled, the Rtm module will be configured with MeasurmentPoints to measure the runtime of the RxIndication and the runtime of the tasks executed during the Sd MainFunction.

Runtime measurement support cannot be used if multi-partition support is enabled (see chapter 5.2.5.5).

5.2.7 Handling of Configuration Options

The SD module provides the possibility to forward received configuration options to the user in order to check and match them against the configured ones. Therefore, a `SdCapabilityRecordMatchCallout` has to be configured as shown in Figure 5-8. A definition of the callout API is given in chapter 4.5.1 <SdCapabilityRecordMatchCalloutName>.

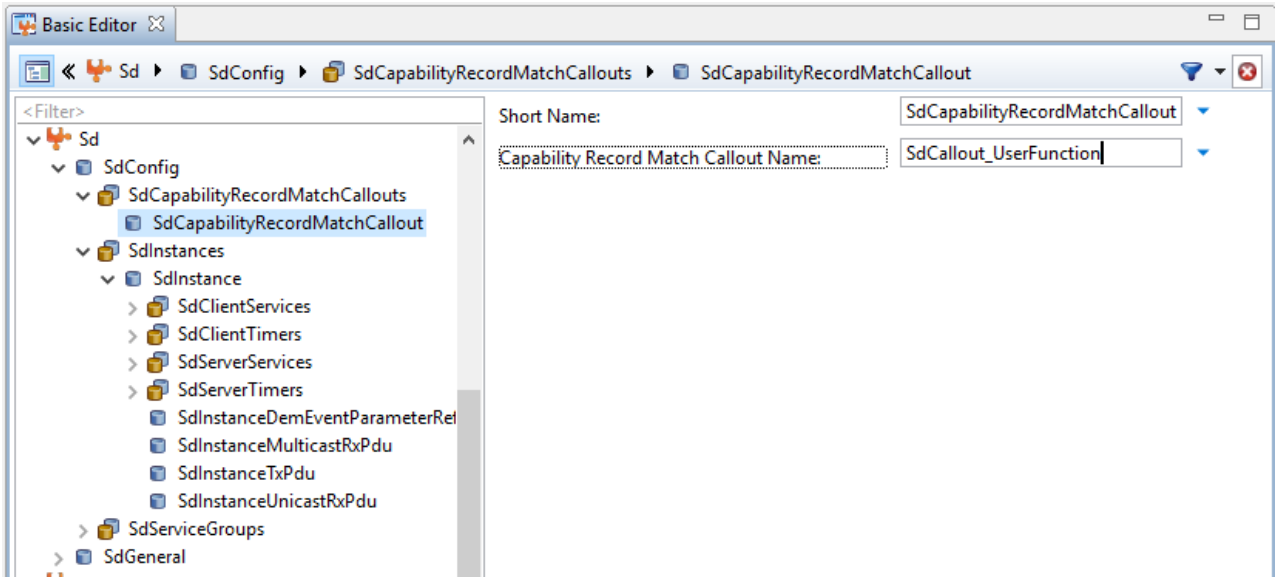

Figure 5-8 Configuration of SdCapabilityRecordMatchCallout

If there is a global `SdCapabilityRecordMatchCallout` configured, the callout can be referenced by any `SdServerService` or `SdClientService`. A configuration of the reference as well as two `SdServerCapabilityRecords` is shown in Figure 5-9.

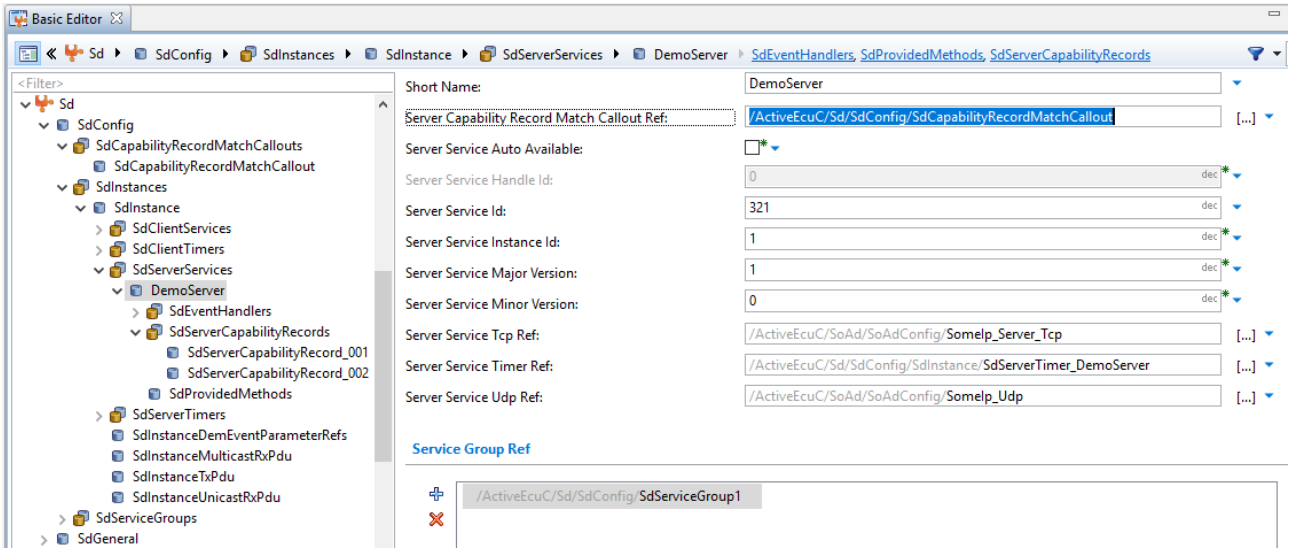

Figure 5-9 Configuration of SdServerCapabilityRecordMatchCalloutRef

The user callout will be invoked if a message for the service is received which contains configuration options or if the service configures `SdServerCapabilityRecords`. The return value of the callout will be used to decide whether the request is valid and shall be processed or shall be ignored (FindService, OfferService, StopOfferService,

StopSubscribeEventgroup, SubscribeEventgroupAck, SubscribeEventgroupNack) or answered negatively (SubscribeEventgroup).

Caution
The user callout is invoked in the (interrupt) context of message reception.

5.2.8 User configuration file

The SD module has an advanced code configuration and code generation tool that completely sets up the module. However, in exceptional cases there is a need to complete or override some of the generated parameters. Most common such cases are workarounds for issues found after product's release.

A user configuration file has no specific name; it can be any text file. In order to use already created user configuration file within the code generation process, you have to specify the full path to this file here:

`/Sd/SdGeneral/SdUserConfigFile`

Caution
User configuration file content must either be described in this manual or agreed to by Vector prior to using it in production code.

5.3 Configuration of Post-Build

The configuration of post-build loadable is described in [7].

The configuration of post-build selectable is described in [8].

For a description which configuration parameters support post-build loadable or selectable please see the Sd_bswmd.arxml file.

6 Glossary and Abbreviations

6.1 Glossary

Term	Description

Table 6-1 Glossary

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
RTE	Runtime Environment
SD	Service Discovery
SOAD	Socket Adaptor
SOMEIP	Scalable service-oriented middleware over IP
SRS	Software Requirement Specification
SwAddrMethods	Software address methods
SWC	Software Component
SWS	Software Specification

Table 6-2 Abbreviations

7 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com