VECTOR

MICROSAR Diagnostic Log and Trace

Technical Reference

Diagnostic Log and Trace

Version 8.1.0

Authors	visml, visore, viszda, vismjv
Status	Released

Document Information

History

Author	Date	Version	Remarks
Visml	2011-06-20	1.0.0	Creation
Visml	2013-03-26	1.0.1	ESCAN00065965: Extended integration directions.
Visore	2013-07-11	1.1.0	ESCAN00068275: AR4-292: Reporting of DET and DEM errors via DLT
Visem	2013-10-07	1.1.1	Typos and formats
Viszda	2015-03-24	1.2.0	Feature: DLT with AUTOSAR functionality and DLT Transport Layer
Viszda	2015-11-20	2.0.0	ESCAN00086655: Create two different Technical References for Dlt_OnXcp and Dlt_Autosar.
Viszda	2016-03-07	2.1.0	Security Mechanism added.
Viszda	2017-07-03	2.1.1	ESCAN00092156
Viszda	2018-11-13	3.0.0	STORY-8329: Add the APIs introduced in ASR DLT 4.3.1.
Viszda	2019-02-06	3.1.0	STORY-9729: Change data types corresponding to ASR 4.3.1 and change the APIs using those data types.
Viszda	2020-03-13	4.0.0	SWAT-15: Support of PduR as lower layer
Viszda	2020-04-21	4.1.0	SWAT-62: store configuration SWAT-78: SWC injection
Viszda	2020-05-28	4.2.0	SWAT-35: Implement the multi log channel support
Viszda	2020-07-09	4.3.0	SWAT-1041: Support of bandwidth management
Viszda	2020-08-05	4.4.0	SWAT-1110: Separate exclusive areas for tx and rx
Viszda	2020-11-25	5.0.0	SWAT-1252: Provide the API to get the default LogLevel and default TraceStatus
Viszda	2021-01-11	5.1.0	SWAT-1070: Validate SoAd parameter to support Dlt over TCP
Visore	2021-11-25	6.0.0	Minor changes
Viszda	2022-08-12	6.1.0	SWAT-2205: support fetching synchronized timestamps from Tsyn; Missing limitation regarding StbM; Missing limitation regarding StbM_GetCurrentTime()
Viszda	2023-02-27	7.0.0	SWAT-2420: Support of snapshot tracing. Added limitation for Dlt protocols. Added limitation for SyncTimestamp service.
Visore	2023-03-29	8.0.0	SWAT-2583: Support DLT AUTOSAR BufferOverflowNotification

Vismjv	2023-10-24	8.0.1	ESCAN00115346 : Wrong description of control message
Viszda	2024-02-20	8.1.0	SWAT-3095: OS Software Tracing ESCAN00116699: Changed response of GetSoftwareVersion

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of Diagnostic Log and Trace	R19-11
[2]	AUTOSAR	Specification of Default Error Tracer	V4.3.1
[3]	AUTOSAR	Specification of Diagnostic Event Manager	V4.2.0
[4]	Vector	UserManual AMD – MICROSAR 4 AUTOSAR Monitoring and Debugging	V1.3.1
[5]	Vector	ASAP2 Tool-Set User Manual	V4.3
[6]	AUTOSAR	Log and Trace Protocol Specification	R19-11
[7]	Vector	TechnicalReference NvM	See delivery
[8]	Vector	TechnicalReference SoAd	See delivery
[9]	Vector	TechnicalReference Os	See delivery

Scope of the Document:

This technical reference describes the use of the DLT module for tracing and logging purposes.

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 DLT as specified in [1].

Supported AUTOSAR Release*:	4	
Supported Configuration Variants:	pre-compile	
Vendor ID:	DLT_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	DLT_MODULE_ID	55 decimal

* For the precise AUTOSAR Release please see the release specific documentation.

AUTOSAR 4.x introduced a new module called DLT which stands for Diagnostic Log and Trace. This module is used to trace messages from DET, DEM and SWCs. According to AUTOSAR the reported messages must be displayed as text messages to the user. This is done by means of a PC tool – in case of Vector it is either CANoe or CANape.

Vector supports the communication protocol specified by AUTOSAR as well as XCP as communication protocol.

1.1 Architecture Overview

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

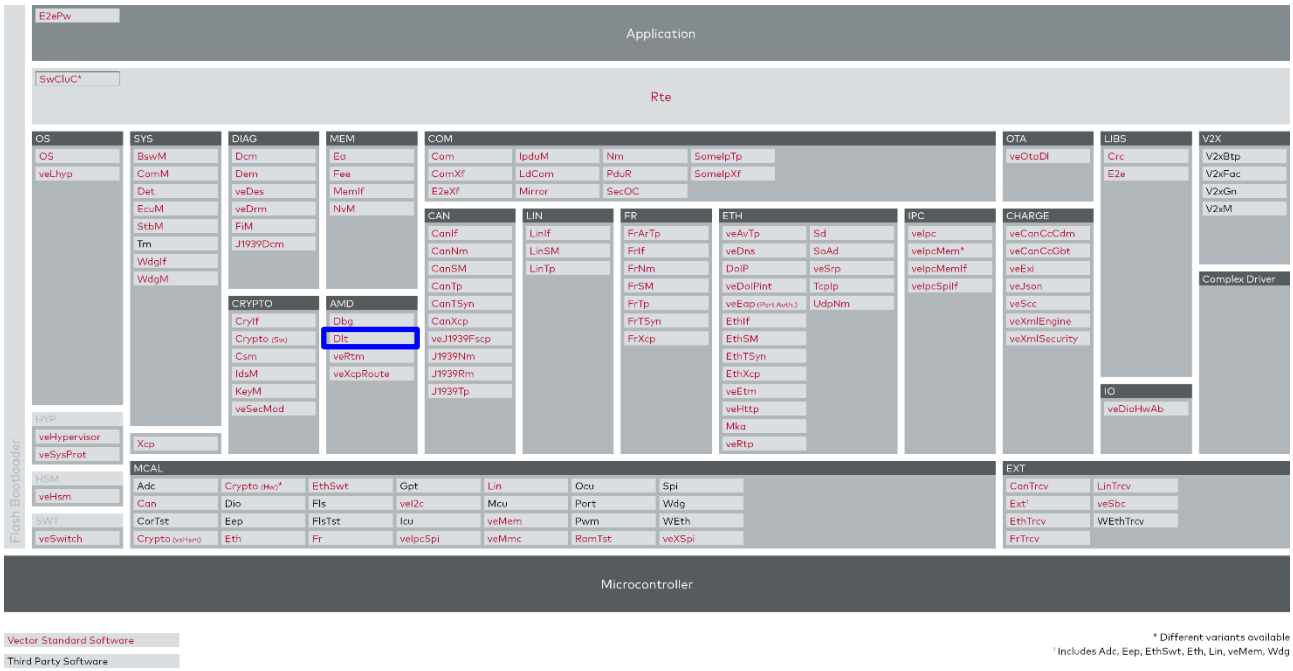


Figure 1-1 AUTOSAR 4.3 Architecture Overview

2 Functional Description

2.1 Features

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

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 Deviations from AUTOSAR standard

2.1.1 Supported features

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Security mechanism: explicit Dlt de-/activation.
Log messages from DET.
Log messages from DEM.
Verbose logging mode.
Non-verbose logging mode.
Logging of errors, warnings, and info messages from AUTOSAR SWCs, providing a standardized AUTOSAR interface.
Gather all log and trace messages from all AUTOSAR SWCs in a centralized AUTOSAR service component (DLT) in BSW.
For DLT on PduR: Runtime configuration of DLT module.
For DLT on PduR: Enable/disable individual log and trace messages.
For DLT on PduR: Client/Server-Port “DltControlService” to change (default) log level, (default) trace status and filter message state. And reset to factory default.
For DLT on PduR: Client/Server-Port “InjectionCallback” to inject application specific services in the application.
For DLT on PduR: Persist the DLT configuration.
For DLT on PduR: Disabling the Rx path.
For DLT on PduR: Multi log channel support.
For DLT on PduR: The control service GetSoftwareVersion (0x13) provides its version in following format: “<MajorVersion>.<MinorVersion>.<PatchVersion>”, where all versions are two ASCII bytes and affected by system endianness.
For DLT on PduR: Bandwidth management (number of bytes to be transmitted is limited in each main function cycle. Therefore, burst transmissions are not possible).
For DLT on PduR: DLT protocol version 1 is supported.

Table 2-1 Supported AUTOSAR standard conform features

2.1.2 Deviations

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

Category	Description
Functional	For DLT on XCP: Runtime configuration of DLT.
Functional	For DLT on XCP: Enable/disable individual log and trace messages.
Functional	For DLT on XCP: Messages for non-verbose logging are stored in an A2L fragment file instead of FIBEX.
Functional	For DLT on XCP: Communication via PduR (instead XCP is used)
API	For DLT on XCP: The APIs Dlt_SendTraceMessage and Dlt_RegisterContext are not provided.
API	For DLT on XCP: The following APIs are not provided: - Dlt_UnregisterContext - Dlt_SetLogLevel - Dlt_SetTraceStatus - Dlt_GetLogInfo - Dlt_GetDefaultLogLevel - Dlt_StoreConfiguration - Dlt_ResetToFactoryDefault - Dlt_SetMessageFiltering - Dlt_SetDefaultLogLevel - Dlt_SetDefaultTraceStatus - Dlt_GetDefaultTraceStatus - Dlt_GetLogChannelNames - Dlt_GetTraceStatus - Dlt_SetLogChannelAssignment - Dlt_SetLogChannelThreshold - Dlt_GetLogChannelThreshold - Dlt_InjectCall_<SESSION>
Functional	For DLT on XCP: Injection calls of SWCs.
Functional	For DLT on XCP: Persistent storage of DLT configuration.
Functional	For DLT on XCP: Rx channel.
Functional	For DLT on XCP: Bandwidth management.
Functional	Communication over standard Dcm channel
Functional	RTE/VFB tracing (must be implemented manually and the hook functions are not generated)
Functional	Timing messages.
Functional	DLT completely event triggered (This DLT implementation uses a main function, therefore it has a receive buffer).
Functional	No support of wildcards in control service GetLogInfo (if given application id or context id are 0, this is identified as invalid).
Functional	Multiple Rx channels.
Functional	For DLT on PduR: All Dlt user (tuple of application id and context id) must be registered before they can send log and trace messages. Messages of unregistered Dlt users are rejected.
Functional	For DLT on PduR: If all log channel assignments of a Dlt user are removed, the messages of this Dlt user are rejected until it is assigned to a log channel again.

Category	Description
Functional	For DLT on PduR: No GetLogInfo response is send when a Dlt user registers to Dlt.
Functional	For DLT on PduR: All DLT protocol versions other than 1 are rejected.
Functional	For DLT on PduR: The StbM cannot be used for timestamps, therefore if ./DltGeneralTimeStampSupport is enabled, also ./DltGeneralGptChannelRef must be available. The reference ./DltGeneralStbMTimeBaseRef is exclusively used for the payload of service "SyncTimestamp" (0x24).
Functional	For DLT on PduR: If ./DltGeneralStbMTimeBaseRef is used, the referenced hardware timer shall not be an OsCounter. Otherwise, the exclusive areas of DLT must still lock all interrupts, but without using the default OS API. For this, set the ./RteExclusiveAreaImplMechanism to 'CUSTOM' and implement the interrupt lock manually. This implementation is hardware dependent, therefore there is no guide how to do this. Therefore, use a free running timer like a GPT.
API	For DLT on PduR: The following control services return `DLT_NOT_SUPPORTED` if requested: - SetComInterfaceStatus - GetComInterfaceStatus - GetComInterfaceNames - SetComInterfaceMaxBandwidth - GetComInterfaceMaxBandwidth - SetTimingPackets - SetVerboseMode - GetVerboseModeStatus - UseTimestamp - GetUseTimestamp - UseExtendedHeader - GetUseExtendedHeader - GetLocalTime
API	For DLT on PduR: The following APIs are not implemented: - DltCom_CancelTransmitRequest - DltCom_SetInterfaceStatus - DltCom_StartOfReception - DltCom_CopyRxData - DltCom_CopyTxData - Dlt_ConditionCheckRead - Dlt_WriteData - Dlt_ReadDataLength - Dlt_ReadData - Dlt_ActivateEvent
API	For DLT on PduR: The APIs Dlt_SendLogMessage, Dlt_SendTraceMessage and Dlt_RegisterContext have return type Std_ReturnType instead of Dlt_ReturnType. Nevertheless, the values of Dlt_ReturnType are returned.
API	For DLT on PduR: Using the IF API of the PDUR is not supported as lower layer of the DLT (only TP).
API	For DLT on PduR: The following API is not implemented:

Category	Description
	- Dlt_UnregisterContext
API	The DLT does not support sending trace and log messages before the module is initialized as specified in AUTOSAR 4.2.2 and later. This means that applications or modules cannot use the DLT APIs Dlt_SendLogMessage and Dlt_SendTraceMessage until the DLT module is fully initialized. On the other hand, it is possible to queue DET (Dlt_DetForwardErrorTrace) and DEM (Dlt_DemTriggerOnEventStatus) events before the DLT module is initialized, as these events are handled separately from the trace and log messages.

Table 2-2 Deviations from AUTOSAR standard

Note
The FIBEX file required for communication via PduR is not generated, thus it has to be created manually.

How to create this file is described in chapter 3.5.8.

2.1.3 Limitations

2.1.3.1 Segmented PDUs

The DLT does not support segmented PDUs, as this is not intended by the DLT protocol!

This means that only complete DLT messages shall be provided in a PDU. It is not allowed to send the first part of a DLT message in one PDU and send the remaining bytes in the next PDU. This applies to Tx and Rx direction.

In Tx direction the DLT takes care to only transmit unsegmented PDUs.

In the Rx direction the DLT cannot handle this, therefore the DLT Client shall take care of transmitting only unsegmented PDUs.

If segmented DLT messages are received, the DLT cannot match the separated parts of DLT message, therefore it rejects the first part. If the second part of DLT message is received in the next PDU, the DLT expects the beginning of a new DLT message and tries to interpret it accordingly, what most likely fails. All following DLT cannot be correctly interpreted as well.

If this happened, only an ECU reset will recover DLT from this state.

This limitation does not apply to streams via TCP.

2.1.4 Additional features

2.1.4.1 Debug mode

The DLT provides different debug modes in which the behavior of buffering and sending varies. There are the following supported debug modes:

1. Continuous_FirstIsBest:
    a. Buffering and sending of log and trace message is always active and simultaneously active.
    b. If the buffer overflows, the new log/trace message is rejected.
    c. Debug events have no impact.
2. OnEvent_FirstIsBest:
    a. Neither buffering nor sending is active initially.
    b. When a debug event is triggered, buffering is started (sending is still inactive).
    c. If the buffer overflows, buffering is stopped and sending is activated.
    d. New debug events are rejected until the buffer is emptied.

Note
The debug mode has no effect on control messages. Control messages can always be buffered and sent.

The debug mode is log channel specific. The initial debug mode can be configured per log channel with the following parameter:

> /MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltLogChannel/DltLogChannelDebugMode

To request a change of a log channel's debug mode during runtime, there is the API Dlt_SetLogChannelDebugMode(). The actual change of debug mode is done in the next call of Dlt_MainFunction().

The current debug mode can be retrieved with the API Dlt_GetLogChannelDebugMode().

A debug event, only required for debug mode 'OnEvent_FirstIsBest', can be triggered with the API Dlt_TriggerLogChannelDebugEvent().

To trigger an initial debug event, before BSW is initialized, DLT provides the following parameter:

> /MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltLogChannel/DltLogChannelInitialDebugEvent

Note
As the debug event API is only available for application and the application runs after initialization of BSW, it is not possible to start buffering before the initialization of BSW is done. To enable this, there is the parameter ‘DltLogChannelInitialDebugEvent’.

This is interesting e.g. for DETs, DEM events, and OS scheduling events occurring before initialization.

2.1.4.2 OS Software Tracing

The DLT provides the possibility to make use of the MSR OS VTHs (timing hooks) to generate and transmit trace messages. These trace messages contain the information of started and/or terminated OS task or category 2 ISR.

Please refer to [9] for more details about the MSR OS VTHs.

The MSR OS provides the following VTHs which are used by DLT for the OS Tracing:

- OS_VTH_SCHEDULE
- OS_VTH_ACTIVATION
- OS_VTH_SETEVENT

2.1.4.2.1 Settings

The DLT defines and reserves some settings for the OS Tracing like the DET and DEM, which are also able to send DLT message triggered by BSW.

OS VTH API and identifiers	OS_VTH_SCHEDULE	OS_VTH_ACTIVATION	OS_VTH_SETEVENT
Dlt API name	Dlt_OsVthSchedule	Dlt_OsVthActivation	Dlt_OsVthSetEvent
Session id (the OS module id)	0x00000001	0x00000001	0x00000001
Application id	‘Cr<CoreIdAs2ASCIILetters>’ (e.g. ‘Cr00’ for core 0)	‘Cr<CoreIdAs2ASCIILetters>’ (e.g. ‘Cr00’ for core 0)	‘Cr<CoreIdAs2ASCIILetters>’ (e.g. ‘Cr00’ for core 0)
Context id	‘OsSh’	‘OsAc’	‘OsSe’
Message id	‘vST’ + 0x01	‘vST’ + 0x01	‘vST’ + 0x01
Message info (MSIN)	0x42 (Non-verbose, DLT_TYPE_APP_TRACE, DLT_TRACE_STATE)	0x42 (Non-verbose, DLT_TYPE_APP_TRACE, DLT_TRACE_STATE)	0x42 (Non-verbose, DLT_TYPE_APP_TRACE, DLT_TRACE_STATE)

Table 2-3 Settings of the OS Tracing

Caution
To be able to use the OS Software Tracing feature, session id 0x00000001, the application ids ‘Cr00’ to ‘Cr99’, and the message ids ‘vST’+0x01 to ‘vST’+0xFF are reserved by DLT.

The used VTHs provide identifiers for the reasons why a thread is activated, terminated, resumed, or preempted. To fit to ARTI, those VTH identifiers are mapped to ARTI identifiers. The mapping is listed in the following Table 2-4.

VTH identifiers	VTH value	ARTI identifiers	ARTI value
-	-	ARTI_OSARTITASK_ACTIVATE	0
OS_VTHP_TASK_ACTIVATION	1	ARTI_OSARTITASK_START	1
OS_VTHP_TASK_WAITEVENT OS_VTHP_TASK_WAITSEMA	4 8	ARTI_OSARTITASK_WAIT	2
-	-	ARTI_OSARTITASK_RELEASE	3
OS_VTHP_THREAD_PREEMPT	16	ARTI_OSARTITASK_PREEMPT	4
OS_VTHP_TASK_TERMINATION	1	ARTI_OSARTITASK_TERMINATE	5
OS_VTHP_THREAD_RESUME	16	ARTI_OSARTITASK_RESUME	6
OS_VTHP_TASK_SETEVENT OS_VTHP_TASK_GOTSEMA	4 8	ARTI_OSARTITASK_CONTINUE	7
OS_VTHP_ISR_START	2	ARTI_OSCAT2ISR_START	16
OS_VTHP_ISR_END	2	ARTI_OSCAT2ISR_STOP	17
-	-	ARTI_OSCAT2ISR_ACTIVATE	18
-	-	ARTI_OSCAT2ISR_PREEMPT	19
-	-	ARTI_OSCAT2ISR_RESUME	20
OS_VTHP_THREAD_CLEANUP	32	-	-

Table 2-4 Mapping of VTH to ARTI identifiers

2.1.4.2.2 OS Tracing messages

The size and content of standard header and the existence of extended header for all OS Tracing messages depend on the configuration:

- /MICROSAR/Dlt/DltConfigSet/DltProtocol/DltHeaderUseEcuId
- /MICROSAR/Dlt/DltConfigSet/DltProtocol/DltHeaderUseSessionID
- /MICROSAR/Dlt/DltConfigSet/DltProtocol/DltHeaderUseTimestamp
- /MICROSAR/Dlt/DltConfigSet/DltProtocol/DltUseExtHeaderInNonVerbMode

Note
The parameter ./DltHeaderUseTimestamp is forced by DLT to be enabled, if OS Tracing is active, as it is considered that tracing messages require a timestamp.

Except of the timestamp, it is recommended to disable all options to reduce the overhead. The information of session id, application id, context id, and MSIN is also provided in the payload.

The payload depends on the VTH and is always transmitted in big endian.

Payload content	Values
Message Id (4 bytes)	‘vST’+0x01 (= 0x76 53 54 01, where 0x01 is the OS Tracing protocol version)
Core id (1 byte)	0x00 (0x00 is the default, but it depends on /MICROSAR/Dlt/DltGeneral/DltAssignedEcucCoreRef)
VTH (1 byte)	0x01 (OS_VTH_SCHEDULE -> Dlt_OsVthSchedule)
FromThreadId (2 byte)	0x0000 – 0xFFFF The task or category 2 ISR identifier.
FromThreadReason (1 byte)	DLT_ARTI_TASK_WAIT (= 0x02) DLT_ARTI_TASK_PREEMPT (= 0x04) DLT_ARTI_TASK_TERMINATE (= 0x05) DLT_ARTI_CAT2ISR_STOP (= 0x11) DLT_ARTI_INVALID (= 0xFF, should never occur, only in case of error)
ToThreadId (2 byte)	0x0000 – 0xFFFF The task or category 2 ISR identifier.
ToThreadReason (1 byte)	DLT_ARTI_TASK_START (= 0x01) DLT_ARTI_TASK_RESUME (= 0x06) DLT_ARTI_TASK_CONTINUE (= 0x07) DLT_ARTI_CAT2ISR_START (= 0x10) DLT_ARTI_INVALID (= 0xFF, should never occur, only in case of error)
EventCounter (1 byte)	0x00 – 0xFF Counts the number of OS Tracing messages (not equal to message counter in the standard header, as this counts all log and trace messages)

Table 2-5 Payload of Dlt_OsVthSchedule

Payload content	Values
Message Id (4 bytes)	‘vST’+0x01 (= 0x76 53 54 01, where 0x01 is the OS Tracing protocol version)
Core id (1 byte)	0x00 (0x00 is the default, but it depends on /MICROSAR/Dlt/DltGeneral/DltAssignedEcucCoreRef)
VTH (1 byte)	0x02 (OS_VTH_ACTIVATION -> Dlt_OsVthActivation)
TaskId (2 byte)	0x0000 – 0xFFFF The task identifier.
Reason (1 byte)	DLT_ARTI_TASK_ACTIVATE (= 0x00)
EventCounter (1 byte)	0x00 – 0xFF Counts the number of OS Tracing messages (not equal to message counter in the standard header, as this counts all log and trace messages)

Table 2-6 Payload of Dlt_OsVthActivation

Payload content	Values
Message Id (4 bytes)	‘vST’+0x01 (= 0x76 53 54 01, where 0x01 is the OS Tracing protocol version)
Core id (1 byte)	0x00 (0x00 is the default, but it depends on /MICROSAR/Dlt/DltGeneral/DltAssignedEcucCoreRef)
VTH (1 byte)	0x03 (OS_VTH_SETEVENT -> Dlt_OsVthSetEvent)
TaskId (2 byte)	0x0000 – 0xFFFF The task identifier.
Reason (1 byte)	DLT_ARTI_TASK_RELEASE (= 0x03)
EventCounter (1 byte)	0x00 – 0xFF Counts the number of OS Tracing messages (not equal to message counter in the standard header, as this counts all log and trace messages)

Table 2-7 Payload of Dlt_OsVthSetEvent

2.1.4.2.3 Configuration

The feature is available when the following parameter is enabled:

> /MICROSAR/Dlt/DltGeneral/DltOsSoftwareTracing

Additionally, it is required to make the DLT using the VTHs. To achieve that, there are two ways:

1. Include the ‘Dlt.h’ in /MICROSAR/Os/OsOS/OsDebug/OsTimingHooksIncludeHeader

2. Include an application header in /MICROSAR/Os/OsOS/OsDebug/OsTimingHooksIncludeHeader

For the first approach, there is nothing else to do, the DLT API is automatically called from corresponding VTHs.

For the second approach, there are some additional steps required in the application header:

1. Implement at least the three VTHs.
2. Add the calls to the corresponding DLT API:
    1. Call Dlt_OsVthSchedule in context of OS_VTH_SCHEDULE.
    2. Call Dlt_OsVthActivation in context of OS_VTH_ACTIVATION.
    3. Call Dlt_OsVthSetEvent in context of OS_VTH_SETEVENT.
3. After/below implementation of the three MSR OS timing hooks, include the ‘Dlt.h’.

2.1.4.2.4 Configuration improvements

If the OS Tracing is enabled, it automatically starts tracing before Dlt_Init() is called. But, per default the global state of DLT changes after Dlt_Init() to ‘OFFLINE. In global state ‘OFFLINE, the OS Tracing is stopped, and no more trace messages are buffered.

To restart the OS Tracing, the application shall call Dlt_SetState() via the corresponding RTE port (‘DLTStateHandling’).

To avoid this gap, set the following parameter to ‘ONLINE’:

> /MICROSAR/Dlt/DltGeneral/DltInitialGlobalState

In debug mode ‘OnEvent_FirstIsBest’ there is another limitation. As a trigger is required to start buffering in this debug mode, which usually only comes from application, the first OS trace events are not buffered.

To avoid this gap, enable the following parameter (at least for the default log channel):

> /MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltLogChannel/DltLogChannelInitialDebugEvent

The DLT does only support single core logging and tracing. If the DLT is used in a multicore system, the DLT uses per default the core 0. To enable the OS Tracing on another core than core 0, there is the following optional reference to the EcuC:

> /MICROSAR/Dlt/DltGeneral/DltAssignedEcucCoreRef

For more details, please refer to following section 2.1.4.3.

2.1.4.3 Explicit core reference

The DLT only supports single core systems but may be integrated in a multi core system. To allow this, the DLT provides the optional reference to an EcuC core:

> /MICROSAR/Dlt/DltGeneral/DltAssignedEcucCoreRef

If this is set, it is expected that all DLT services (e.g. Dlt_InitMemory(), Dlt_Init(), Dlt_MainFunction(), ..) are exclusively called on this ECU core.

The only exceptions are the following APIs which request the currently active core from OS and only execute the service if the core is equal to the configured core:

> Dlt_OsVthSchedule()

> Dlt_OsVthActivation()

> Dlt_OsVthSetEvent()

> Dlt_DetForwardErrorTrace()

> Dlt_DemTriggerOnEventStatus()

> Dlt_SendLogMessage()

> Dlt_SendTraceMessage()

> Dlt_GetVersionInfo()

Caution
Even if the reference (./DltAssignedEcucCoreRef) is set, the DLT still only supports single core. Therefore, logging and tracing is only possible on the selected core.

This also means that DETs can only be forwarded on this core, all other DETs are ignored.

That all services (except of the services mentioned above) are exclusively be called on the referenced core shall be validated by the integrator!

2.2 Initialization

The DLT module is pre-initialized by a call to function Dlt_InitMemory(). The initialization includes all global data required to log data from ECU start on.

The DLT module is initialized by a call to the function Dlt_Init(). Initialization of the DLT must not be done before the XCP driver is initialized and the corresponding bus interfaces and drivers. The function has a pointer parameter which is not used in the current version. Therefore, the user should pass a NULL pointer.

2.3 States

The DLT provides two global states:

1. DLT_GLOBAL_STATE_OFFLINE
2. DLT_GLOBAL_STATE_ONLINE

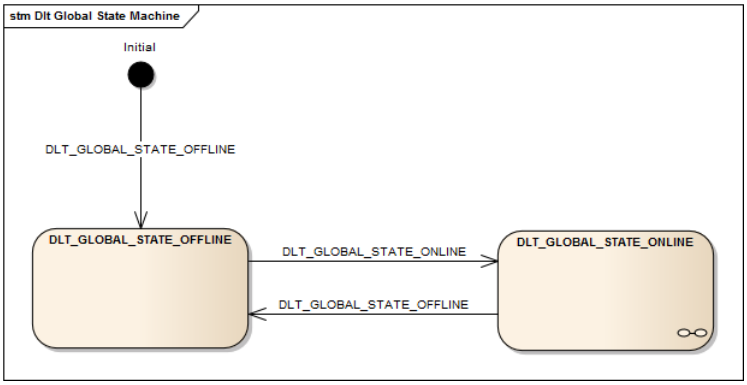


Figure 2-1 Global state machine with default configuration

The default initial global after Dlt_Init() is “OFFLINE”, this can be changed with the following parameter:

> /MICROSAR/Dlt/DltGeneral/DltInitialGlobalState

In the global state “OFFLINE”, most services are block (buffering, sending, and receiving messages).

In the global state “ONLINE”, all services are allowed.

As the DLT provides services to change and request ECU internal data, it is recommended to use the default setting (“OFFLINE”) if there is an Rx PDU (/MICROSAR/Dlt/DltGeneral/DltGeneralRxDataPathSupport is enabled) or for production.

Note
For the OS Tracing, it is recommended to set the initial global state to ‘ONLINE’, to trace all OS scheduling events immediately after initialization.

With the API Dlt_SetState() a state change can be requested. The next call to Dlt_MainFunction() changes the state as requested.

The DLT on PduR supports a sub state machine within the global state DLT_GLOBAL_STATE_ONLINE. It is shown in the following Figure 2-2.

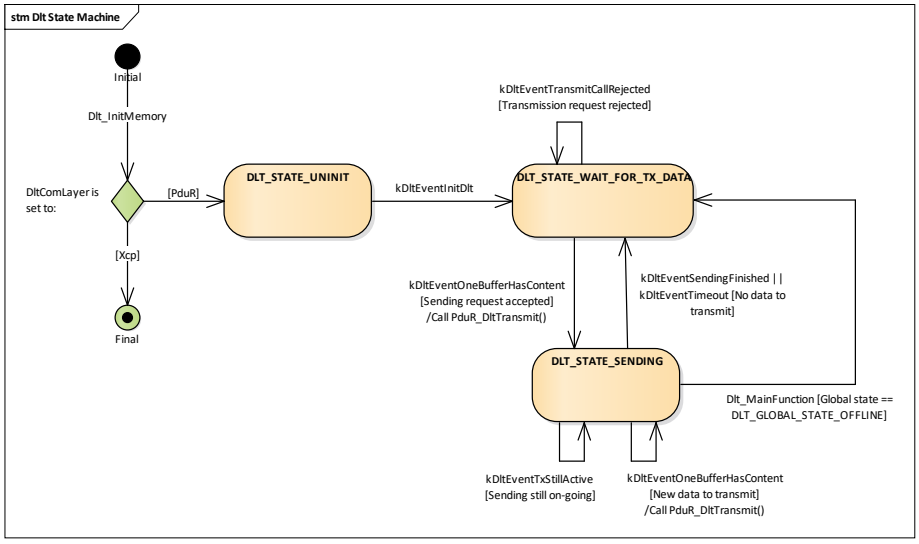


Figure 2-2 Dlt state machine

2.4 Main Functions

In case of XCP as protocol layer, the main function only handles the global state machine.

In case of DLT on PduR the Dlt_MainFunction additionally controls the sub state machine of DLT.

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 [2], if development error reporting is enabled (i.e. pre-compile parameter `DLT_DEV_ERROR_REPORT==STD_ON`).

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 DLT ID is 55.

The reported service IDs identify the services which are described in 4.2. The following table presents the service IDs and the related services:

Service ID	Service
0x01	Dlt_Init
0x02	Dlt_GetVersionInfo
0x03	Dlt_SendLogMessage
0x04	Dlt_SendTraceMessage
0x05	Dlt_RegisterContext
0x06	Dlt_ResetToFactoryDefault
0x07	Dlt_DetForwardErrorTrace
0x08	Dlt_SetLogLevel

Service ID	Service
0x09	Dlt_SetTraceStatus
0x0A	Dlt_GetLogInfo
0x11	Dlt_SetDefaultLogLevel
0x12	Dlt_SetDefaultTraceStatus
0x14	Dlt_InjectCall_<SessionId>
0x15	Dlt_DemTriggerOnEventStatus
0x17	Dlt_GetLogChannelNames
0x1A	Dlt_StoreConfiguration
0x1B	Dlt_SetMessageFiltering
0x1F	Dlt_GetTraceStatus
0x20	Dlt_SetLogChannelAssignment
0x21	Dlt_SetLogChannelThreshold
0x22	Dlt_GetLogChannelThreshold
0x80	Dlt_SetDebugMode
0x81	Dlt_GetDebugMode
0x82	Dlt_TriggerDebugEvent
0x83	Dlt_OsVthSchedule
0x84	Dlt_OsVthActivation
0x85	Dlt_OsVthSetEvent
0x50	Dlt_MainFunction
0x51	Dlt_SetState
0x52	Dlt_GetState

Table 2-8 Service IDs

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

Error Code		Description
0x01	DLT_E_PARAM	API service called with wrong parameter.
0x02	DLT_E_PARAM_POINTER	API service called with a NULL pointer. In case of this error, the API service shall return immediately without any further action, besides reporting this development error.
0x03	DLT_E_INIT_FAILED	API was unable to initialize the service.
0x04	DLT_E_REGISTRATION	Too many contexts are registered with the DLT module. (e.g. the configuration tables are full)
0x05	DLT_E_SKIPPED_TRANSMISSION	Runtime error: Message could not be sent.
0x06	DLT_E_DEPRECATED_PARAMETER	Runtime error: A deprecated parameter with a value different to 0 for a Dlt command has been received.
0x07	DLT_E_MULTIPLE_REQUESTS	Runtime error: Multiple Control Requests at the same time.

Error Code		Description
0x10	DLT_E_UNINITIALIZED	The DLT is not initialized, thus the requested Service is not available.
0x11	DLT_E_INVALID_STATE	The DLT is in an invalid global state.

Table 2-9 Errors reported to DET

2.5.2 Production Code Error Reporting

The DLT module does not report Production Code Errors to the DEM.

3 Integration

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

3.1 Embedded Implementation

The delivery of the DLT consists out of these files:

File Name	Description	Integration Tasks
Dlt.c	This is the source file of the DLT module.	-
Dlt.h	This is the header file of the DLT module.	-
Dlt_Types.h	Defines required data types which are not defined by RTE.	-
Dlt_Cbk.h	Callback header file containing the callback declarations for the PduR (if PduR is active).	-
Dlt_Cfg.h	Generated header file for pre-compile time configuration data.	-
Dlt_Lcfg.c	Generated link time configuration source file.	-
Dlt_Lcfg.h	Generated link time configuration header file.	-

Table 3-1 Implementation Files

3.2 Critical Sections

The DLT module uses three exclusive areas as described in the following Table 3-2.

Exclusive Area	Description
DLT_EXCLUSIVE_AREA_APIS	Protects the computation of the reported message code and state transitions in all Dlt APIs. DLT on XCP: Additionally, this exclusive area protects the function Dlt_DetForwardErrorTrace from interruption by itself. Since it is called from all possible contexts in the BSW a global lock is the safest way of configuration.
DLT_EXCLUSIVE_AREA_TX	DLT on PduR: This exclusive area protects the access of Tx buffer during execution Dlt_CopyTxData. Because the Dlt_MainFunction accesses the Tx buffer as well, it is recommended to implement this exclusive area as global lock as well. If Dlt_CopyTxData is called in context of Dlt_MainFunction, this exclusive area may be defined empty. DLT on XCP: This exclusive area is not used.

Exclusive Area	Description
DLT_EXCLUSIVE_AREA_RX	DLT on PduR: This exclusive area protects the access of Rx buffer during execution Dlt_StartOfReception and Dlt_CopyRxData. Because the Dlt_MainFunction accesses the Rx buffer as well, it is recommended to implement this exclusive area as global lock as well. DLT on XCP: This exclusive area is not used.

Table 3-2 Exclusive Areas of the DLT module

3.3 Common Configuration Steps

The DLT provides services to change and request ECU internal data. To avoid misuse of these services, the DLT provides a security mechanism. After initialization (in default setting, please refer to 2.3) the DLT is in state `DLT_GLOBAL_STATE_OFFLINE`, where most services, like communication to external client, are not available.

The activation of DLT has to be requested explicitly via a call of `Dlt_SetState(DLT_GLOBAL_STATE_ONLINE)`. With next call of `Dlt_MainFunction` all services of DLT are available. To grant no misuse, the call of `Dlt_SetState` should be in context of a diagnostic session.

With a call of `Dlt_SetState(DLT_GLOBAL_STATE_OFFLINE)` the DLT is deactivated again. All stored messages are removed from DLT buffers.

These APIs shall be called from the application. Therefore, the RTE provides interfaces for the application to access these APIs indirectly.

Caution
It is highly recommended to call the APIs `Dlt_SetState` and `Dlt_GetState` in context of a diagnostic session.

3.4 DLT on XCP

There are two versions of DLT for AUTOSAR 4; DLT on XCP and AUTOSAR DLT. The difference is the communication layer used and the feature coverage. DLT on XCP uses XCP as communication protocol and supports a subset of features.

3.4.1 XCP Event

The DLT module uses XCP for data transmission to the PC tool which provides a GUI to display the reported messages.

There are four XCP events that can be triggered by DLT. Table 3-3 shows these events and the functions where they are triggered.

Function	XCP event
Dlt_SendLogMessage	Dlt_EvtMsg Dlt_EvtVMsg
Dlt_DemTriggerOnEventStatus	Dlt_EvtDem
Dlt_DetForwardErrorTrace	Dlt_EvtDet

Table 3-3 DLT functions and their corresponding XCP events

Each time one of the functions in Table 3-3 is called the DLT module calls the corresponding XCP event to trigger transmission of the message.

Note
Your XCP driver needs to be configured in a way that XCP events are used.

3.4.2 XCP: Logging of verbose and non-verbose DLT messages

The DLT supports the verbose and non-verbose transmission mode for log messages.

> Non-Verbose Mode:

In the Non-Verbose Mode the XCP master measures only the Message ID. This Message ID is unique for a specific DLT message and the static message text is associated to this ID. The mapping between Message ID and static text is provided by the generated A2L file.

Example
Non-verbose DLT messages can be transmitted as following:

```
Dlt_NonVerboseMsgType nonVerboseMsg = { {0, 0, 0 }, DltConf_DltNonVerboseMessage_Msg1};
Dlt_MessageLogInfoType msgLogInfoType = { DLT_LOG_ERROR, DLT_NON_VERBOSE_MSG, 0, 0};

retVal = Dlt_SendLogMessage( 0, &msgLogInfoType, (P2CONST(uint8, AUTOMATIC,
DLT_APPL_VAR)) &nonVerboseMsg, 0 );
```

Expert Knowledge
Instead of using an integer value as Message ID when calling Dlt_SendLogMessage() you can use the generated Symbolic Name Value define like in the example above. The generated define depends on the short name of the corresponding DltNonVerboseMessage container (marked red in the example).

This way the identification of static messages is easier.

The corresponding static message can be configured as depicted below:

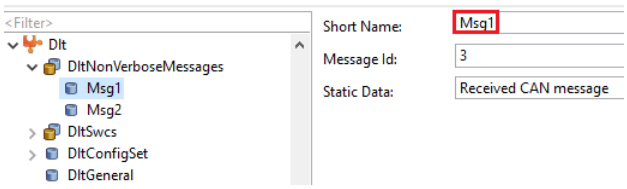


Figure 3-1 Configuration of non-verbose messages

> **Verbose Mode:**

The Verbose Mode is supported if DLT_USE_VERBOSE_MODE is STD_ON (`/MICROSAR/Dlt/DltConfigSet/DltProtocol/DltUseVerboseMode`). In contrary to the Non-Verbose Mode, XCP master measures and displays the complete non-static text in the Verbose Mode. The maximum allowed message length is defined by the preprocessor switch DLT_MAX_MESSAGE_LENGTH (`/MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltLogChannel/DltLogChannelMaxMessageLength`).

Example

Verbose DLT messages can be transmitted as following:

```
Dlt_VerboseMsgType verboseMsg = { {0, 0, 0 }, 3 /* MessageId */, "Test Message"};
Dlt_MessageLogInfoType msgLogInfoType = { DLT_LOG_INFO, DLT_VERBOSE_MSG, 0, 0};


retVal = Dlt_SendLogMessage( 0, &msgLogInfoType, (P2CONST(uint8, AUTOMATIC,
DLT_APPL_VAR)) &verboseMsg, sizeof("Test Message\n"));
```

or using sprintf():

```
Dlt_VerboseMsgType verboseMsg = { {0, 0, 0 }, 3 /* MessageId */};
Dlt_MessageLogInfoType msgLogInfoType = { DLT_LOG_INFO, DLT_VERBOSE_MSG, 0, 0};
uint16 len = sprintf ( verboseMsg.Payload, "The half of %d is %d", 60, 60/2 );
retVal = Dlt_SendLogMessage( 0, &msgLogInfoType, (P2CONST(uint8, AUTOMATIC,
DLT_APPL_VAR)) &verboseMsg, len);
```

Caution

The DLT_MAX_MESSAGE_LENGTH parameter must be configured with caution because this parameter directly affects the runtime of the Dlt_SendLogMessage() function when logging verbose messages.

3.4.3 Workflow McData

To use any AMD module (DBG, DLT or RTM) with XCP as protocol layer, XCP requires the mapping between symbols and memory addresses. For this purpose, there is a master a2l

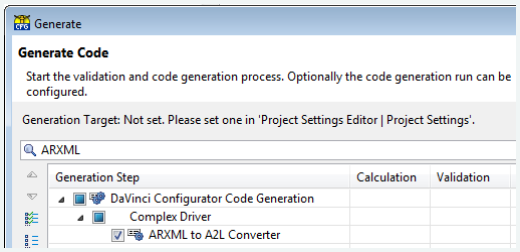
file. This a2l file can then be used by CANoe or CANape to stimulate and/or acquire data from ECU.

Practical Procedure

1. Copy .\external\Misc\McData_master.a2l to .\Config\McData\ a rename it to master.a2l.

2. Open it with an editor and edit all lines with “TODO” as required for your purpose.

3. Open your config with DaVinci Configurator. There is a generator called “ARXML to A2L Converter”. This generator has no configurable content, it just has to be generated.



This converter creates the McData.a2l and McDataExport.arxml. These files contain all data required for the symbols which will later be available in CANoe.

4. Switch to .\CANoe and create the init file Updater.ini if it does not exist. Open it in an editor and add following lines:

```
[ELF]
ELF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=100

[UBROF]
UBROF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=70

[OMF]
OMF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=70

[PDB]
PDB_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=70

[OPTIONS]
FILTER_MODE=1
MAP_FORMAT=31  ;Refer to ASAP Updater User Manual for
description of MAP file format
```

The bold values should be individually adapted. The MAP_MAX_ARRAY defines how big the maximum array can be that is read/written by XCP.

The MAP_FORMAT has to be set to according to the used map file. For example set it to 31 if you use an elf file (32bit CPU). Set it to 54 if you are using CANoe.Emu. For other values please refer to [5].

If your project is compiled the a2l updater has to be executed. If you have no updater yet, create a batch file called update_a2l.bat in the directory where your map file is. Open it with an editor. The content should include the absolute path to the ASAP2Updater.exe, and the relative paths to:

- the input file (Master.a2l)
- the output file (AmdResult.a2l)
- the map file (<ProjectName>.<[elf|pdb|map|…]>)
- the Update.ini file
- and optionally an log file (AsapUpdater.log)

For example this could look like:

"C:\Program Files (x86)\Vector CANwin 8.5\ASAP2Updater\Exec\ASAP2Updater.exe" -I ..\Config\McData_Master.a2l -A .\Tsp_BAC4_REF_VC121.elf -O ..\Canoe\AmdResult.a2l -T ..\Canoe\Updater.ini -L ..\Log\AsapUpdater.log

The updater uses the Master a2l, containing the symbols, and the map file to map all symbols to memory addresses.

XCP can now use the result a2l.

3.4.4 DLT Reporting with CANoe

Note
You need a license of CANoe's XCP option in order to use it as a XCP master

CANoe provides no map updater. Therefore, it is necessary to run an address update of the A2L files using external updater tools (like Vector's ASAP Toolset). Afterwards you can load the resulting A2L file into CANoe by selecting XCP/CCP Configuration in the CANoe's Configuration menu.

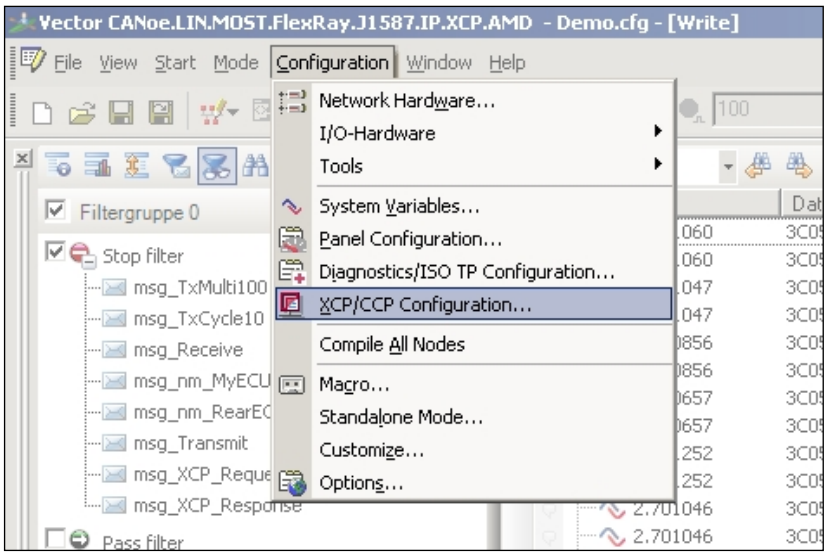


Figure 3-2 Open XCP/CCP option

- Add the master A2L which includes the DLT A2L fragments via the [Add…] button to the XCP configuration.
- DLT Reporting of DET Errors: Select the “Signal Configuration” tab and add the signal Dlt_DetErrorCode to your measurement list. Select DLT_DET as XCP event channel for measurement as shown in Figure 3-3.
- DLT Reporting of DEM Event Status Changes: Select the “Signal Configuration” tab and add the signal Dlt_DemEventStatus to your measurement list. Select DLT_DEM as XCP event channel for measurement.
- DLT Logging of Non-Verbose Messages: Select the “Signal Configuration” tab and add the signal Dlt_NonVerboseMessageId to your measurement list. Select DLT_MSG as XCP event channel for measurement.
- DLT Logging of Verbose Messages: Select the “Signal Configuration” tab and add the signal Dlt_VerboseMessageData to your measurement list. Select DLT_VMSG as XCP event channel for measurement.

Note
DLT Logging of Verbose Messages requires at least CANoe 8.1 SP3 otherwise the measured data will be displayed as raw value instead of string.

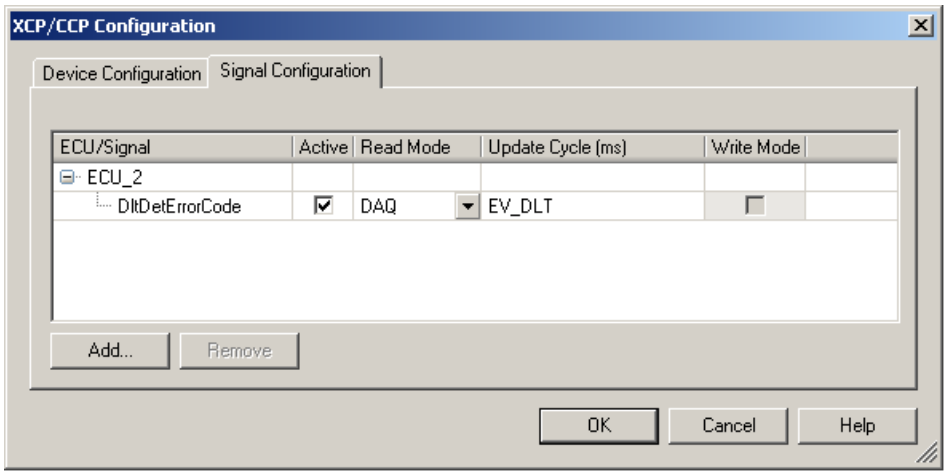


Figure 3-3 Selection of the XCP measurement object for DLT logging of DET reports

In order to display the reported DLT messages you can add a new trace window to your CANoe configuration and filter out all messages and events except for example Dlt_DetErrorCode. It is important to select the [sym] button in the toolbar in order to enable the output of textual DLT messages into CANoe’s trace window.

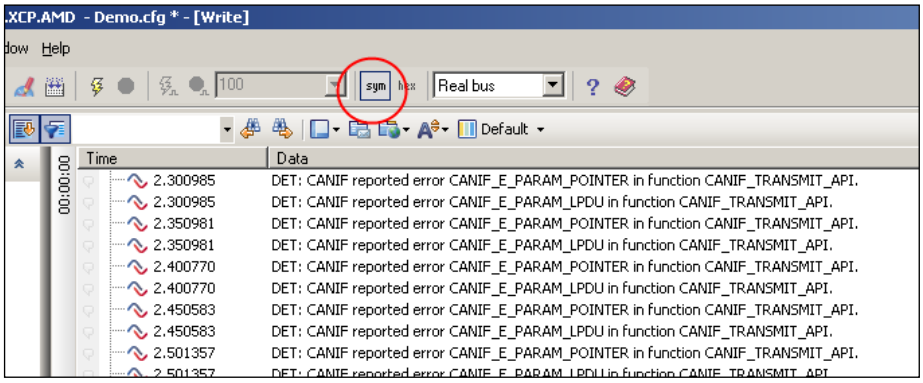


Figure 3-4 Enable the symbolic option by clicking the button [sym] in CANoe's toolbar.

Note

The raw values (32bit in hex) of reported DET errors can be interpreted as following:

0x II : Instance ID
MM : Module ID
SS : Service ID
EE : Error ID

E.g: 0x00380202 /* Instance 0x00,
Module ID = 0x38 -> "SoAd",
Service ID = 0x02 -> "SoAd_GetVersionInfo",
Error ID = 0x02 -> "SOAD_E_PARAM_POINTER" */

3.4.5 DLT Reporting with CANape

In order to log DLT messages with CANape the file Dlt.A2L has to be included into a master A2L file. The addresses in the A2L file can be updated using CANape's map updater. Once the A2L file is incorporated into the CANape configuration one can select the DLT measurement object using CANape's symbol explorer. The DLT variable for measurement can be found in the DLT folder in the A2L file as shown in Figure 3-5.

- DLT Reporting of DET Errors: Select the variable Dlt_DetErrorCode and add it to a text window as shown in Figure 3-6.
- DLT Reporting of DEM Event Status Changes: Select the variable Dlt_DemEventStatus and add it to a text window.
- DLT Logging of Non-Verbose Messages: Select the variable Dlt_NonVerboseMessageId and add it to a text window.
- DLT Logging of Verbose Messages: Select the variable Dlt_VerboseMessageData and add it to a text window.

DLT messages can be displayed using CANape's text window (Figure 3-7).

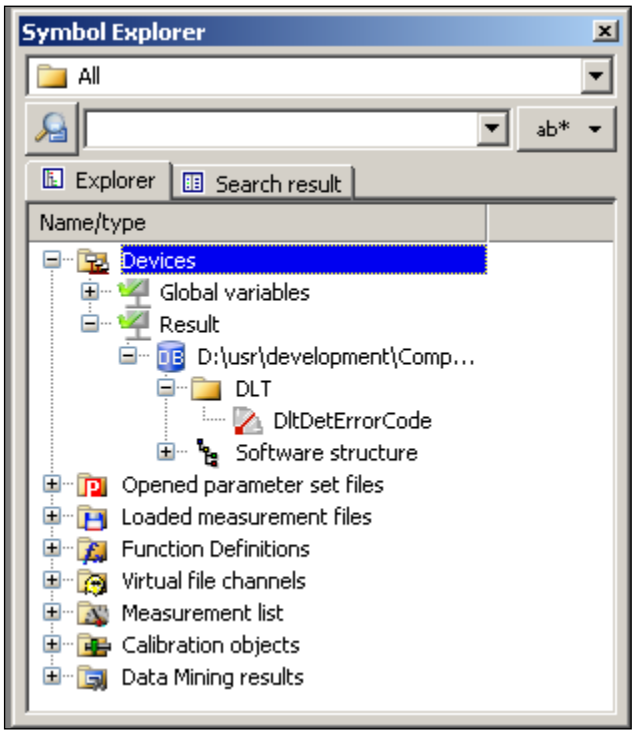


Figure 3-5 CANape's symbol explorer for selection of the DLT variable

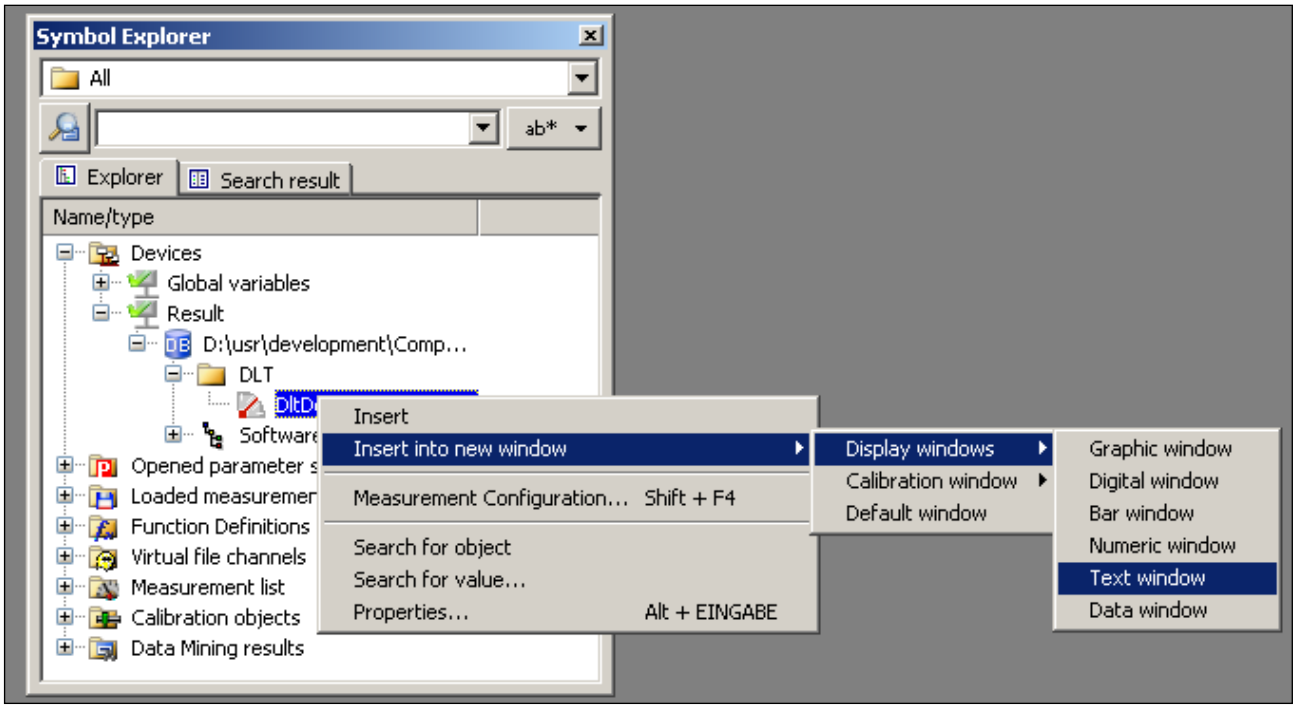


Figure 3-6 Select the DLT variable for measurement in a text window

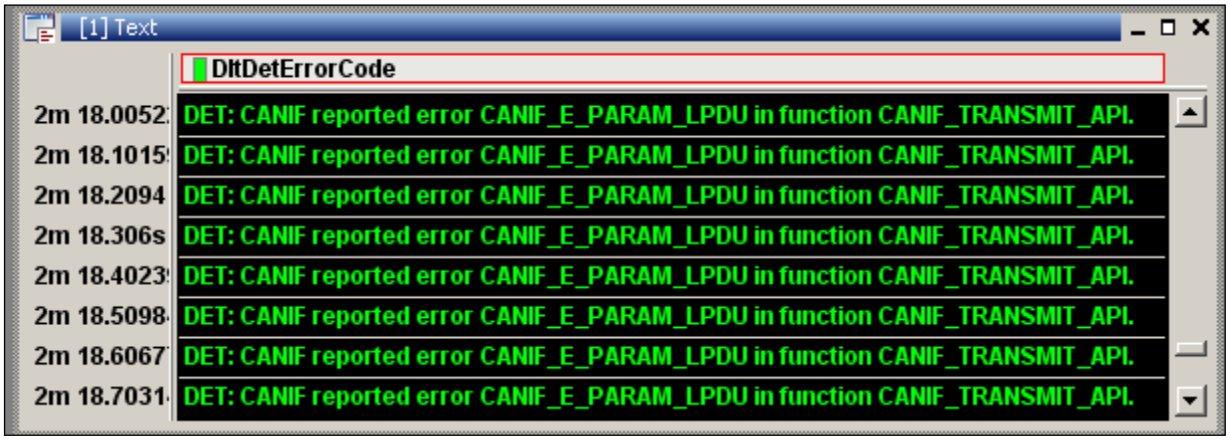


Figure 3-7 Logging of DLT messages using CANape's text window.

3.5 AUTOSAR DLT (on PduR)

The DLT implemented as specified by AUTOSAR uses the PduR as communication layer. The PduR enables the DLT to send its messages via different busses (e.g. the SoAd).

This feature is licensed and not part of the standard DLT module of Vector.

3.5.1 Configuration

3.5.1.1 DaVinci Configurator Pro

3.5.1.1.1 EcuC

Create two PDUs in `/MICROSAR/EcuC/EcucPduCollection/Pdu` for Tx and 2 PDUs for Rx (if required). Call them (for example):

- Pdu_Dlt_Tx_DltToPduR
- Pdu_Dlt_Tx_PduRToSoAd
- Pdu_Dlt_Rx_PduRToDlt
- Pdu_Dlt_Rx_SoAdToPduR

Set PduLength as required.

3.5.1.1.2 DLT

To use the DLT as specified in AUTOSAR `/MICROSAR/Dlt/DltGeneral/DltComLayer` must be set to "PduR".

The use of PduR is licensed, therefore the correct license must be available. Otherwise, the generation of DLT is prohibited.

Dlt SWCs:

The DLT SWCs represent an SWC using the DLT:

- `/MICROSAR/Dlt/DltSwc`

Each SWC has a session id and the option to be notified about log level threshold and trace status changes:

> ./DltSwcSessionId

> ./DltSwcSupportLogLevelChangeNotification

Note
Only configured SWCs have the possibility to be informed about threshold and trace status changes.
Therefore, SWCs registering at runtime cannot be notified.

a) DLT SWC contexts

The DLT provides the possibility to configure explicit contexts for each SWC:

> ./DltSwcContext

A configured context is automatically registered to the DLT and therefore does not need to call the API Dlt_RegisterContext at runtime. A context consists of an application id and a context id; this tuple identifies a DLT user.

> ./DltSwcApplicationId

> ./DltSwcContextId

Note
The same application id may be used with different context ids, but an application id must be used exclusively by one SWC context.

DLT config set:

The DLT config set provides a set of configuration variants, like Tx- and Rx-PDUs, explicit log level thresholds and trace status assignment.

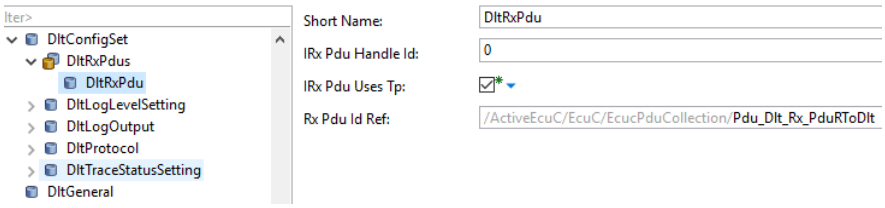
> /MICROSAR/Dlt/DltConfigSet

a) Rx PDU:

The Rx PDU is optional, and does only exists if /MICROSAR/Dlt/DltGeneral/DltGeneralRxDataPathSupport is checked:

> ./DltRxPdu

Set its parameter like in the following figure (with correct PDU).



Caution
The DLT only supports the TP interfaces of the PduR.

b) Log channel(s):

The log channel represents the interface from DLT to PduR.

> `./DltLogOutput/DltLogChannel`

Note
The DLT supports one or more log channels.

For each log channel the following parameters can be configured:

> `./DltLogChannelBufferSize`

> `./DltLogChannelId`

> `./DltLogChannelMaxMessageLength`

> `./DltLogChannelThreshold`

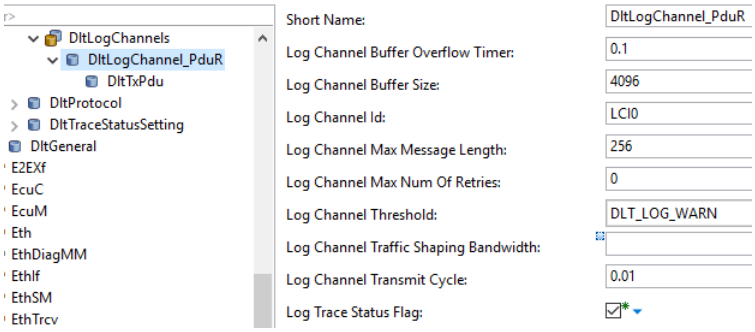
> `./DltLogTraceStatusFlag`

> `./DltLogChannelTrafficShapingBandwidth`

Note
The traffic shaping bandwidth parameter is given as bit/s. This is internally re-calculated to byte/MainFunctionCycle. Therefore, only this number of bytes can be transmitted in one MainFunctionCycle.

The Dlt only transmits complete messages, therefore it is possible that not all available bytes are used for transmission. But the not transmitted bytes are not added for transmission in the MainFunctionCycle.

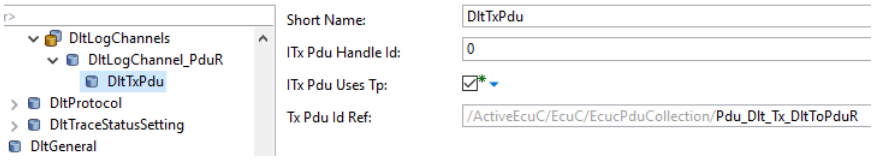
All other parameters are not supported.



Additionally, a log channel has a Tx PDU:

> `./DltTxPdu`

Set its parameter like in the following figure (with correct PDU).



c) Assignments:

The DLT provides the possibility to assign each SWC context the following parameters explicitly:

> `./DltLogLevelSetting/DltLogLevelThreshold`

> `./DltLogOutput/DltLogChannelAssignment`

> `./DltTraceStatusSetting/DltTraceStatusAssignment`

If there is no explicit assignment to a SWC context, the corresponding default value is used instead.

> `./DltLogLevelSetting/DltDefaultLogLevel`

> `./DltLogOutput/DltDefaultLogChannelRef`

> `./DltTraceStatusSetting/DltDefaultTraceStatus`

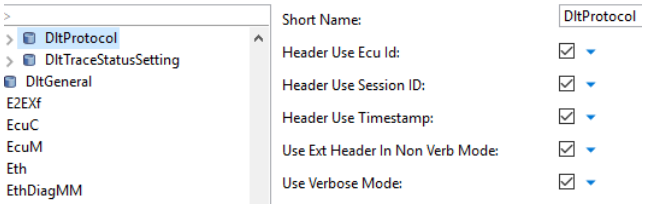
Note
Only one threshold and only one trace status may be assigned to an SWC context.

d) DLT protocol:

The DLT message can be extended by optional information which can be chosen during configuration:

> ./DltProtocol

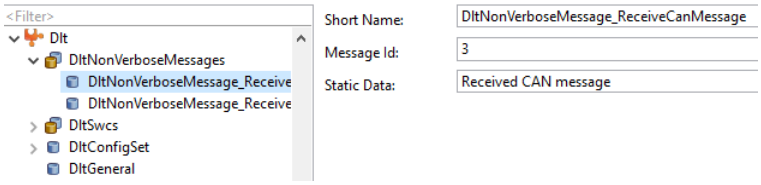
The optional information is added at runtime to the DLT message header:



NonVerboseMessages:

The DLT provides the possibility to generate some message ids which can be used by SWCs (if Dlt_Cfg.h or Dlt.h is included).

> /MICROSAR/Dlt/DltNonVerboseMessage



The message id is calculated automatically during calculation step of the DaVinci Configurator 5.

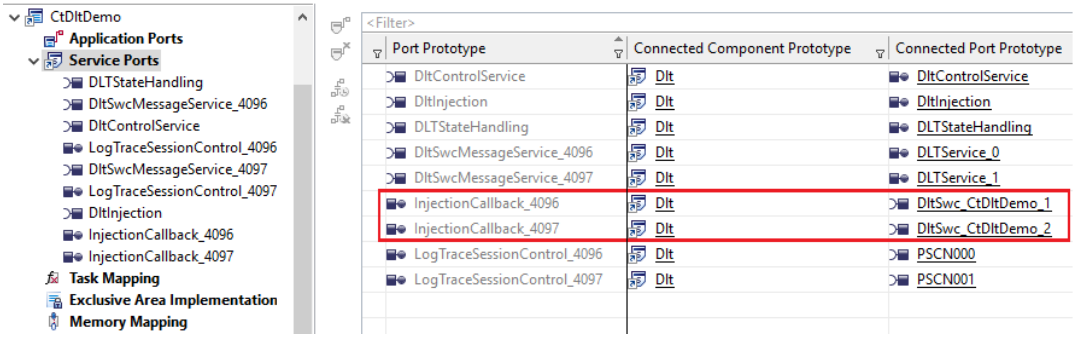
SWC Injection:

To enable the SWC injection, set the following parameter:

> /MICROSAR/Dlt/DltGeneral/DltGeneralInjectionSupport

Note
This is a global configuration. Therefore, if it is enabled each configured DltSwc provides a client-/server-port InjectionCallback. It is not possible to decide this for single DltSwc.

In the following example, the SWC CtDltDemo provides two Sessions (4096 and 4097) which are connected to the configured DltSwcs (DltSwc_CtDltDemo_1 and DltSwc_CtDltDemo_2). So, CtDltDemo receives the injection request and executes the service if applicable.

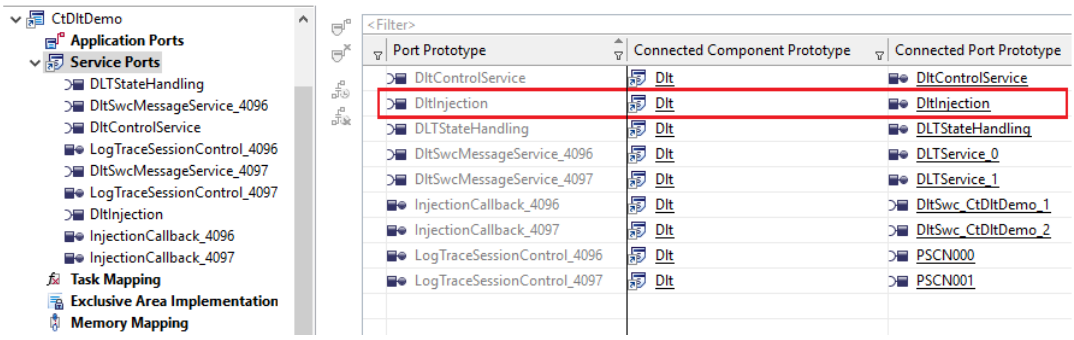


The DLT SWC injection can be requested by application and DLT master.

a) By Application:

To inject a service in a specific SWC, the application (another or the same SWC) has to connect to the client-/server-port `DltInjection`.

In the following example, the SWC `CtDltDemo` is connected to the `DltInjection`.



b) By DLT master:

To allow the DLT master to inject any service to an SWC, the following parameter must be additionally enabled:

> `/MICROSAR/Dlt/DltGeneral/DltGeneralRxDataPathSupport`

Store configuration:

To store the current Dlt configuration, the following parameter must be enabled:

> `/MICROSAR/Dlt/DltGeneral/DltGeneralNvRAMSupport`

Additionally, a reference to a NvM block must be set:

> `/MICROSAR/Dlt/DltGeneral/DltGeneralNvRamRef`

For the configuration of the referenced NvM block, please refer to section 3.5.1.1.4.

3.5.1.1.3 PduR

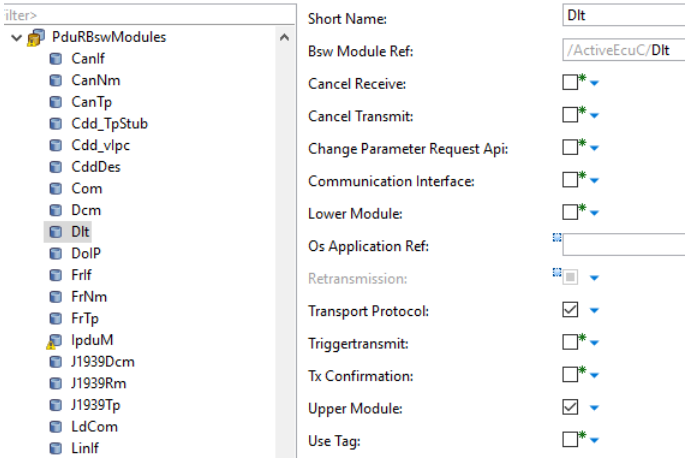
Add the DLT as Bsw module of the PduR:

> /MICROSAR/PduR/PduRBswModules

And check the following parameter:

> ./PduRTransportProtocol

> ./PduRUpperModule



Add a PduR routing path for the Tx and optionally for the Rx path:

> /MICROSAR/PduR/PduRRoutingTables/PduRRoutingTable/PduRRoutingPath

Tx path:

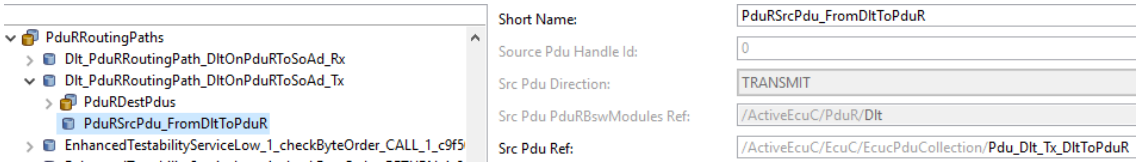
a) Source PDU:

> ./PduRSrcPdu

Refer the PDU between DLT and PduR:

> ./PduRSrcPduRef

➔ Pdu_Dlt_Tx_DltToPduR



b) Destination PDU:

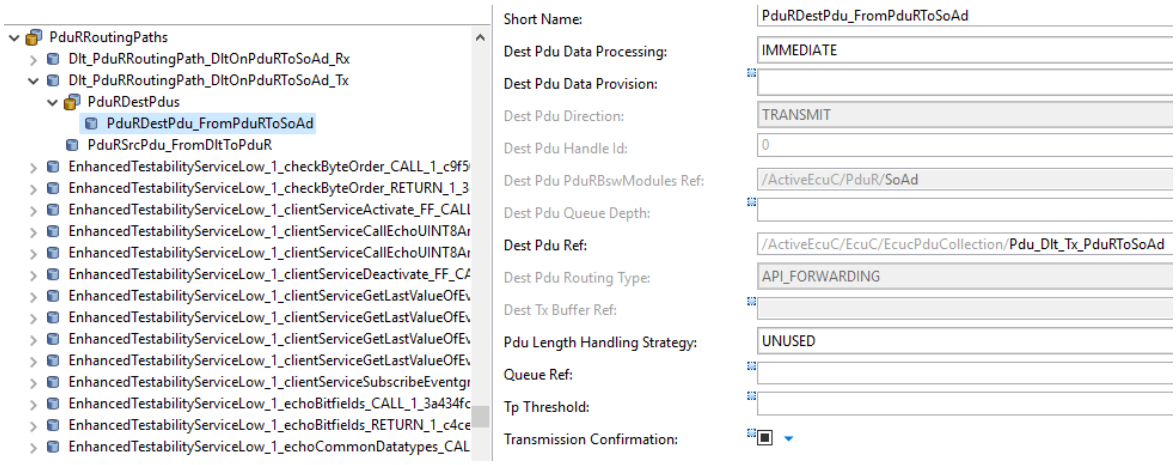
> ./PduRDestPdu

Refer the PDU between PduR and (e.g.) SoAd:

> `./PduRDestPdu/PduRDestPduRef`

➔ Pdu_Dlt_Tx_PduRToSoAd

The other settings should be derived automatically.



Rx path:

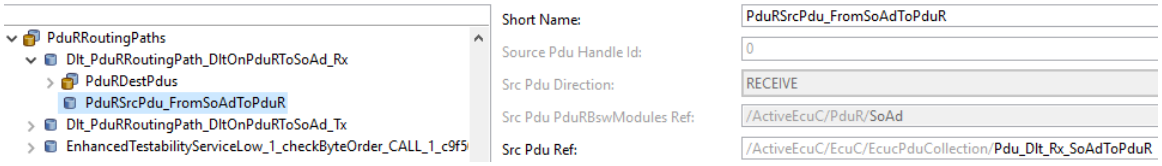
a) Source PDU:

> `./PduRSrcPdu`

Refer the PDU between (e.g.) SoAd and PduR:

> `./PduRSrcPduRef`

➔ Pdu_Dlt_Rx_SoAdToPduR



b) Destination PDU:

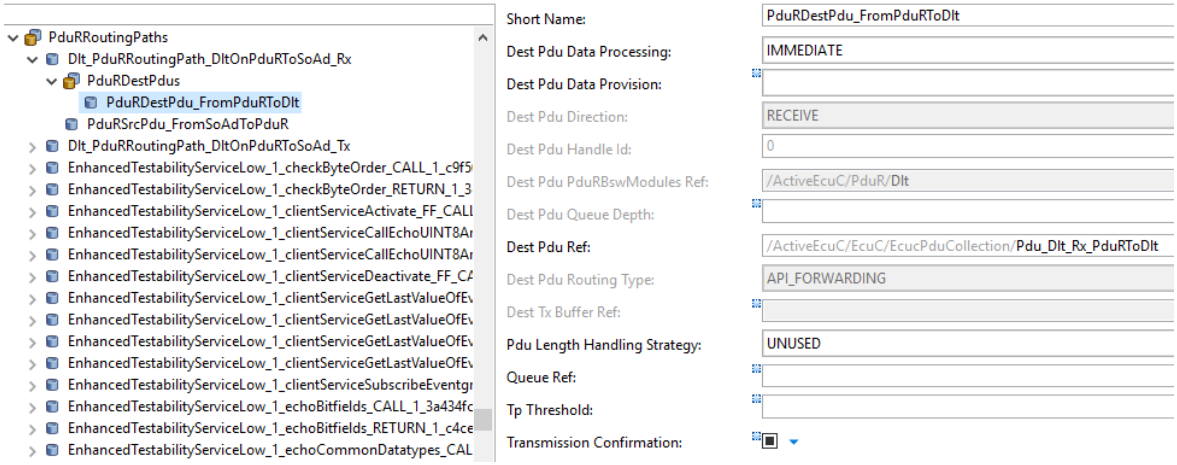
> `./PduRDestPdu`

Refer the PDU between PduR and DLT:

> `./PduRDestPdu/PduRDestPduRef`

➔ Pdu_Dlt_Rx_PduRToDlt

The other settings should be derived automatically.



3.5.1.1.4 NvM

The configuration of the referenced NvM block is shown in the following table.

Parameter	Value/description
NvMNvBlockLength	Value is automatically calculated via validator and depends on number of DltSwc, and DltLogChannel, and DltOsSoftwareTracing.
NvMBlockLengthCheck	It is recommended to set this value (checks that NvM block is greater or equal to RAM representation).
NvMBlockLengthCheckType	Optional to make the length check strict (the configured length must be exactly correct).
NvMRamBlockDataAddress	Must be set to “Dlt_NvData”.
NvMResistantToChangedSw	Depends on changed number of DltSwc and DltLogChannel. If not changed since last call to NvM_WriteBlock: enable this feature. If changed, disable this feature. Please refer to the [7].
NvMSelectBlockForReadAll	This must be enabled, because the DLT does not call NvM_ReadBlock in Dlt_Init. This would lead to an asynchronous initialization. Instead, the block is read during NvM_ReadAll.
NvMSelectBlockForWriteAll	This must be disabled, because the changed DLT configuration must only be persisted if explicitly requested by application or DLT master.

Table 3-4 Configuration of NvM block referenced by DLT

3.5.1.1.5 SoAd

Dlt is independent of the SoAd, therefore UDP and TCP is support for communication, but there is one exception.

When the MICROSAR SoAd and TCP shall be used, the following parameter must exist, and it must be enabled:

> /SoAd/SoAdConfig/SoAdSocketConnectionGroup/SoAdSocketAutomaticSoConSetupKeepOnline

Note
The parameter ‘SoAdSocketAutomaticSoConSetupKeepOnline’ has the effect, that the socket connection is not closed automatically by the SoAd. Therefore, the socket connection must be closed by …
1. The Dlt master (e.g. DltViewer) or
2. a timeout, configured within the SoAd
   -> Enable the parameter following:
> /SoAd/SoAdConfig/SoAdSocketConnectionGroup/SoAdSocketProtocol/SoAdSocketTcp/SoAdSocketTcpKeepAlive

For more information about timeouts, please refer to [8].

3.5.1.2 DaVinci Developer

The DLT provides services that can be used by the application. If a SWC wants to use these services, it must provide other services itself.

3.5.1.2.1 All ports provided and used by Dlt

Services provided by DLT:

> On **Client**/Server port “DltSwcMessageService”:
>> Dlt_RegisterContext
>> Dlt_SendLogMessage
>> Dlt_SendTraceMessge
> On **Client**/Server port “DltControlService”:
>> Dlt_SetLogLevel
>> Dlt_GetLogInfo
>> Dlt_SetTraceStatus
>> Dlt_GetTraceStatus
>> Dlt_SetDefaultLogLevel

- Dlt_SetDefaultTraceStatus
- Dlt_SetMessageFiltering
- Dlt_ResetToFactoryDefault
- Dlt_StoreConfiguration
- Dlt_GetLogChannelNames
- Dlt_GetLogChannelThreshold
- Dlt_SetLogChannelThreshold
- Dlt_SetLogChannelAssignment

- On **Client**/Server port “DLTStateHandling”:
  - Dlt_SetState
  - Dlt_GetState
- On **Client**/Server port “DltInjection”:
  - Dlt_InjectCall_<SessionId>

Services provided by SWC:

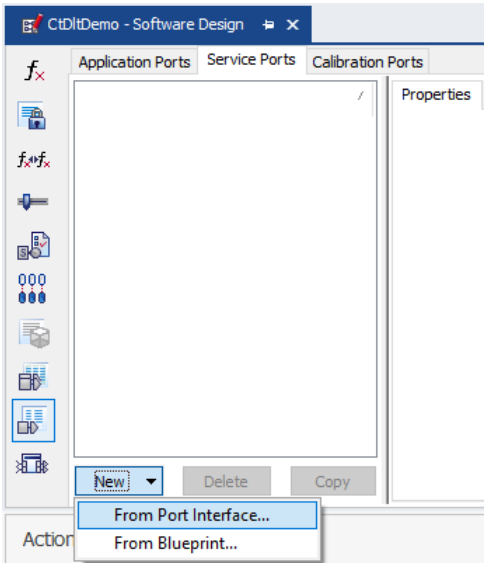
- On Client/**Server** port “LogTraceSessionControl”:
  - LogLevelChangedNotification
  - TraceStatusChangedNotification
- On Client/**Server** port “InjectionCallback”:
  - InjectCall

3.5.1.2.2 Activate the DLT

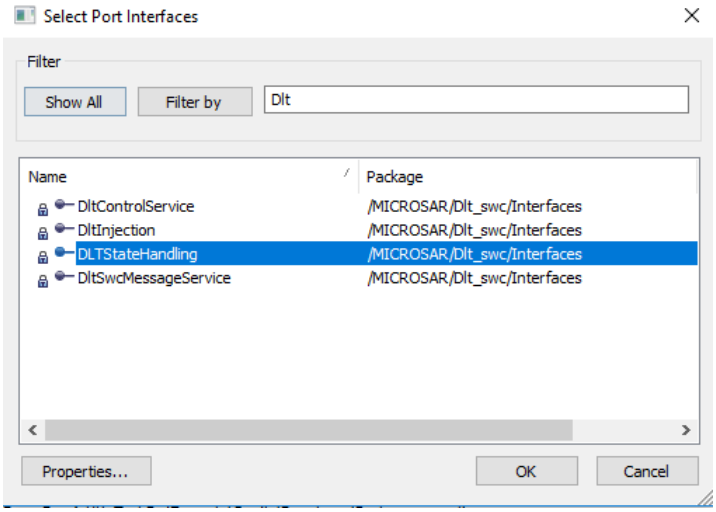
Initially the DLT will reject requests from SWC to introduce as little as possible runtime. If the DLT must be used, it has to be activated first.

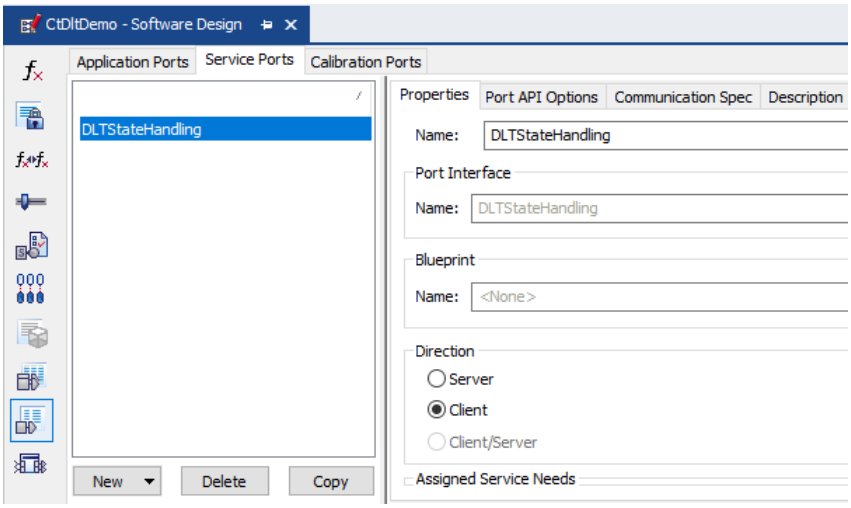
To activate the DLT, the DLT provides the service port ‘DLTStateHandling’.

To add this service port, open the SWC (here the example SWC ‘CtDltDemo’ is used) which should use the DLT service in the DaVinci Developer. Click ‘Port Prototype List’, ‘New’ and ‘From Port Interface…’.

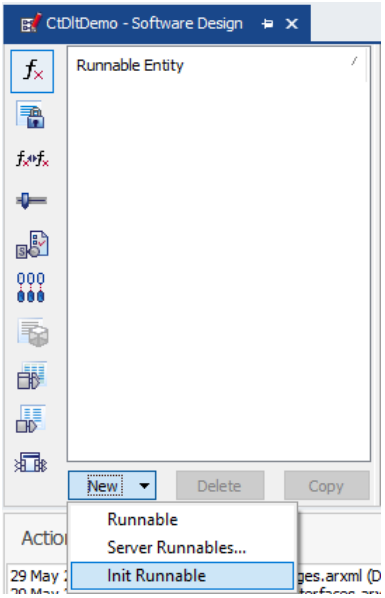


Filter by ‘Dlt’, select ‘DLTStateHandling’ and click ‘OK’.

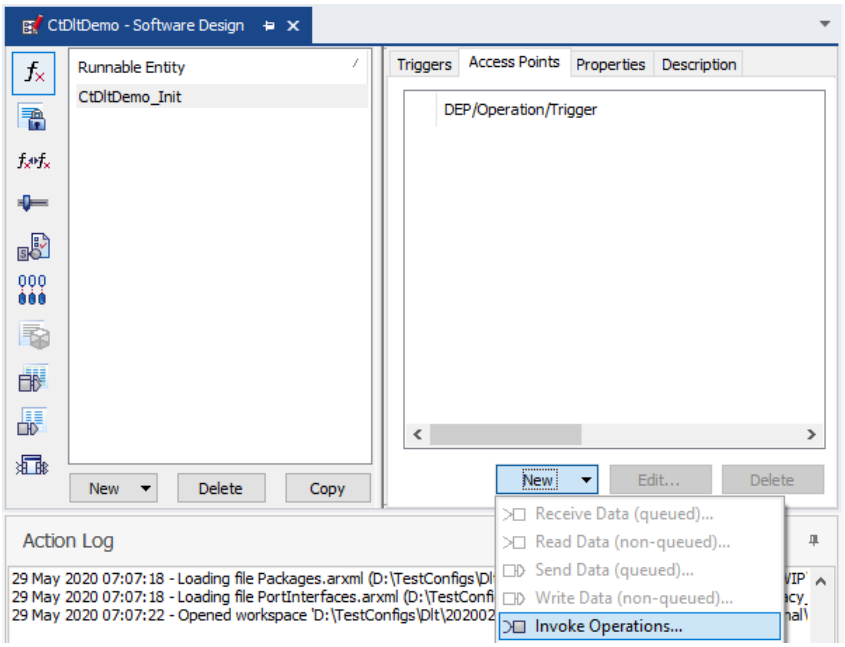




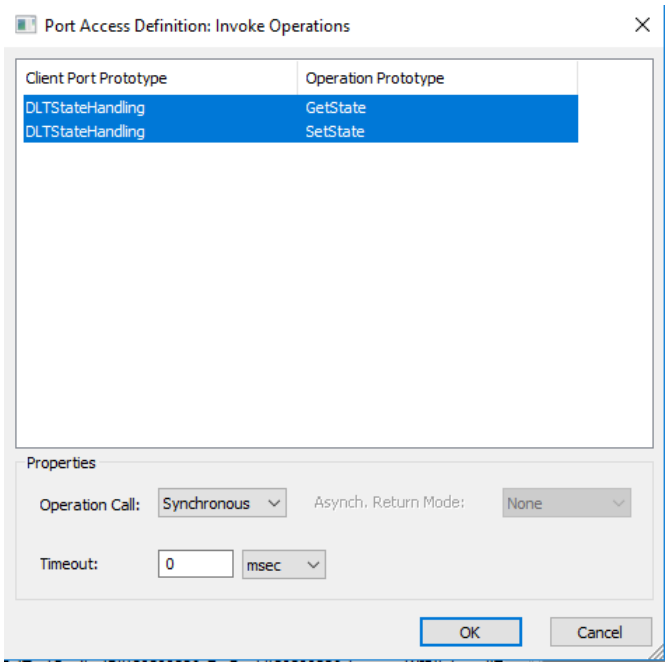
The port must be used by a runnable. Because, the DLT must only be activated once, it is OK to start it from an init runnable. Therefore, click ‘Runnable Entity List’, ‘New’, and ‘Init Runnable’.



Now we have the init runnable ‘CtDltDemo_Init’. Select this runnable, click ‘Access Points’, ‘New’, and ‘Invoke Operations…’.



Select both ‘Operation Prototypes’ and click ‘OK’.

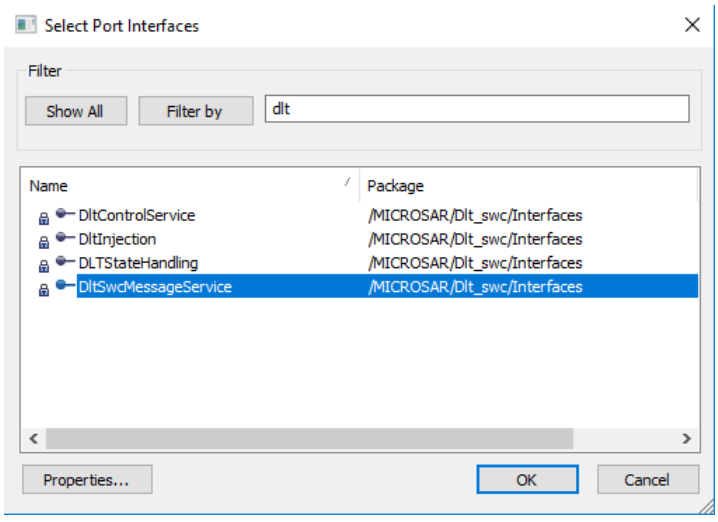


Now it is possible to (de-)active the DLT during runtime.

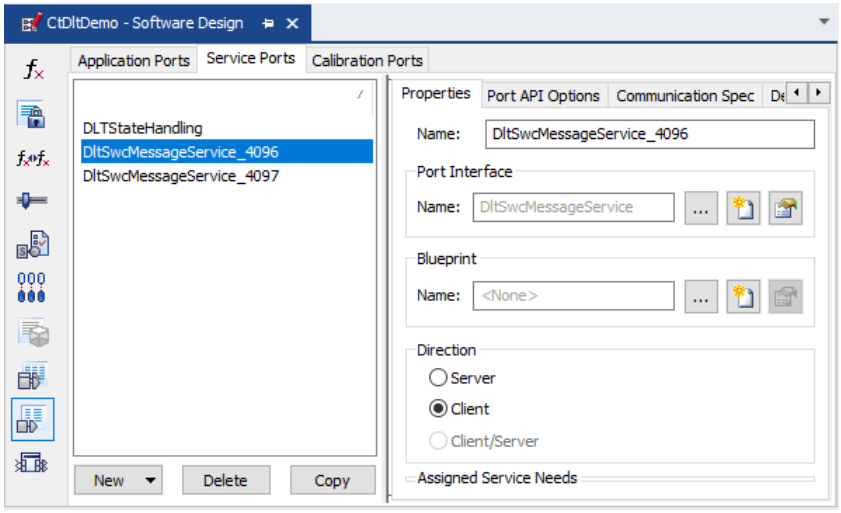
3.5.1.2.3 Sending log and trace messages

The main purpose of DLT is to send log and trace messages. A SWC can add the service port ‘DltSwcMessageService’ to be become able to send log and trace messages.

Therefore, add another service port from port interface (please refer to 3.5.1.2.2). Filter by ‘Dlt’ and select the port interface ‘DltSwcMessageService’ and click ‘OK’.

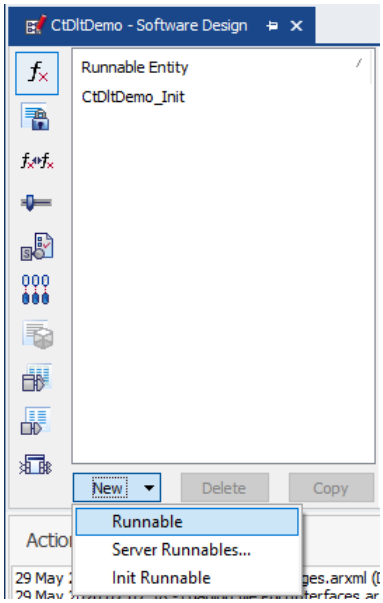


In this example, we create two `/MICROSAR/Dlt/DltSwc` which are both implemented by the SWC ‘CtDltDemo’, therefore we add another ‘DltSwcMessageService’ and rename both like ‘DltSwcMessageService_4096’ and ‘DltSwcMessageService_4097’.

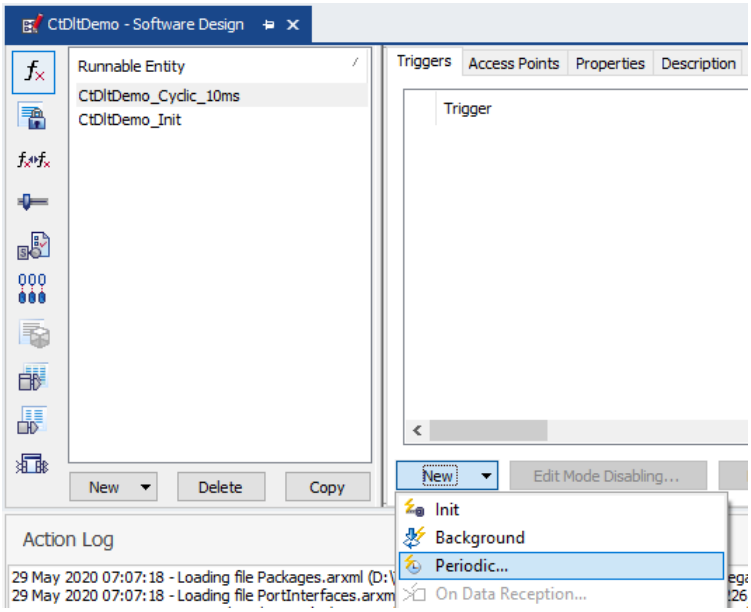


In this example, we want to send cyclically, therefore we add a cyclically called runnable which accesses the new ports.

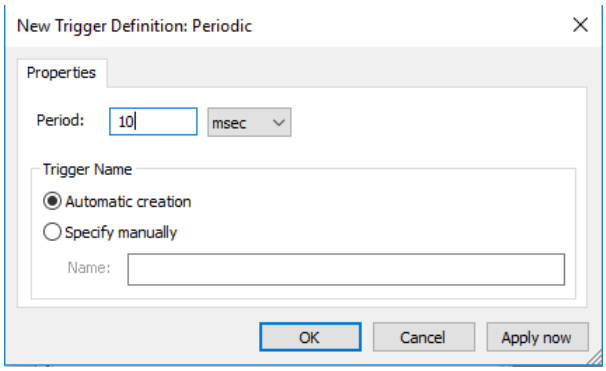
Therefore, click ‘Runnable Entity List’, ‘New’, and ‘Runnable’.



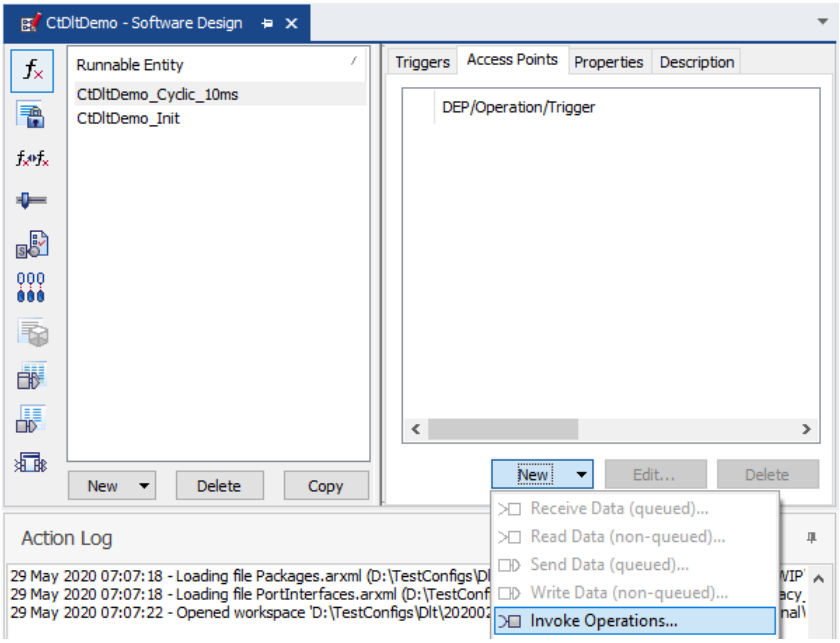
In this example, we call the runnable ‘CtDltdemo_Cyclic_10ms’ and add a periodic trigger. Therefore, select the new runnable, click ‘Triggers’, ‘New’, and ‘Periodic…’.



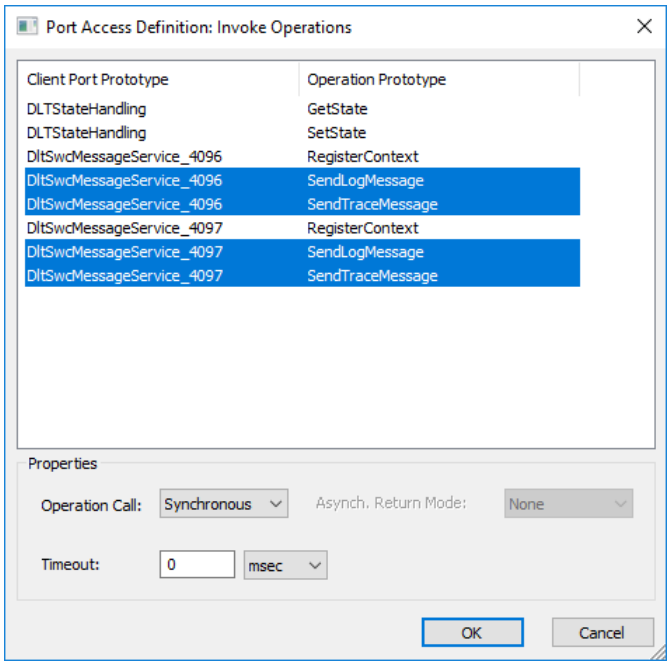
Change ‘Period’ to 10 msec and click ‘OK’.



Now we have to add the port services of ‘DltSwcMessageService_4096’ and ‘DltSwcMessageService_4097’. Therefore, click ‘Access Points’, ‘New’, ‘Invoke Operations…’.



Select the operation prototypes ‘SendLogMessage’ and ‘SendTraceMessage’ of both ports and click ‘OK’.



Now the CtDltDemo is able to send log and trace messages.

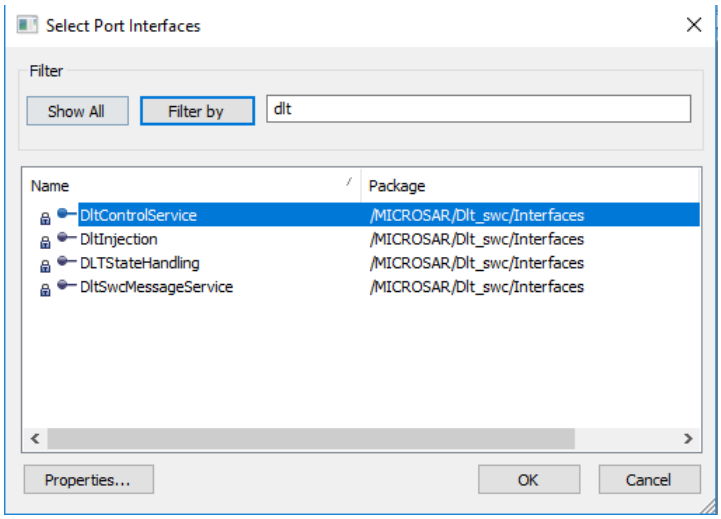
Note that there are two operation prototypes ‘RegisterContext’ which we do not use. They could be added by the init runnable to register new contexts, but in this example, we only use configured `/MICROSAR/Dlt/DltSwc/DltSwcContext`. The configured contexts do not need to be registered during runtime, therefore we do not need those prototypes.

For this topic, please refer to section 3.5.6.1.

3.5.1.2.4 Controlling the DLT settings

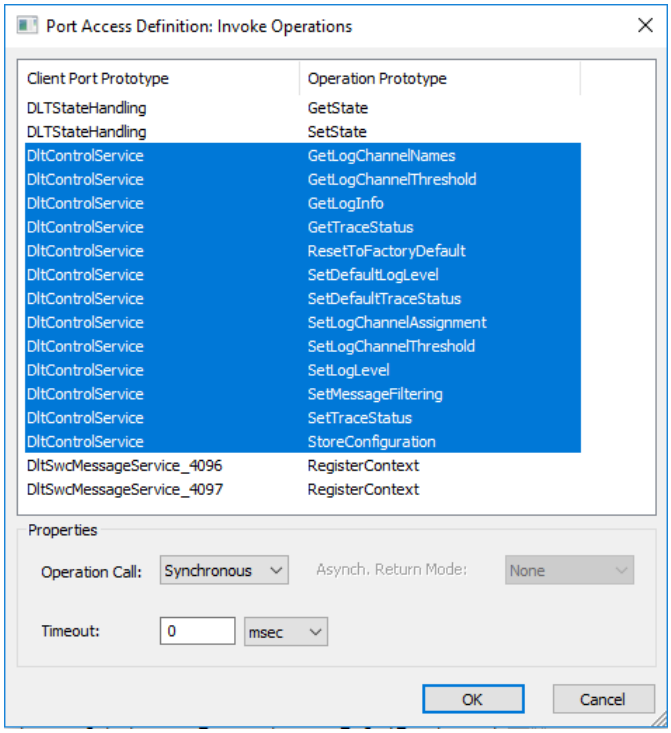
The DLT provides the possibility to change and get some DLT settings, therefore the DLT provides the service port ‘DltControlService’.

Therefore, add another service port from port interface (please refer to 3.5.1.2.2). Filter by ‘Dlt’ and select the port interface ‘DltControlService’ and click ‘OK’.



In this example, we extend the existing runnable ‘CtDltDemo_Cyclic_10ms’ by the operations of ‘DltControlService’. Therefore, we add more invoked operations (for more details please refer to section 3.5.1.2.3).

Select all required operations (in this example, all operations are added) and click ‘OK’.

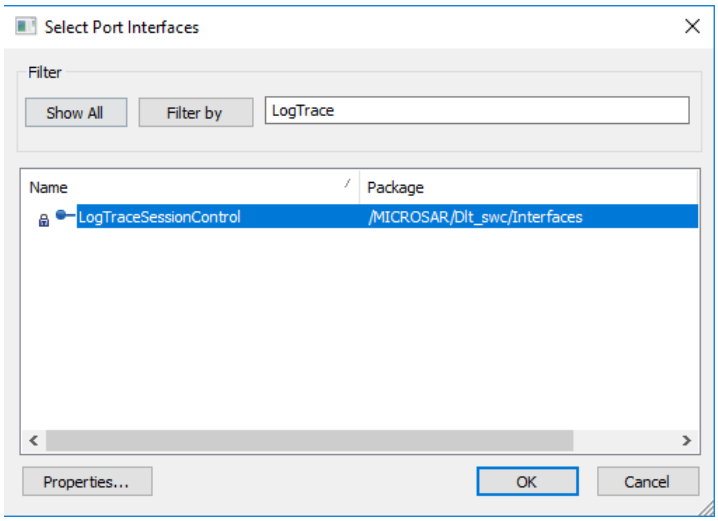


Now it is possible to get and change DLT settings during runtime.

3.5.1.2.5 Threshold change notification

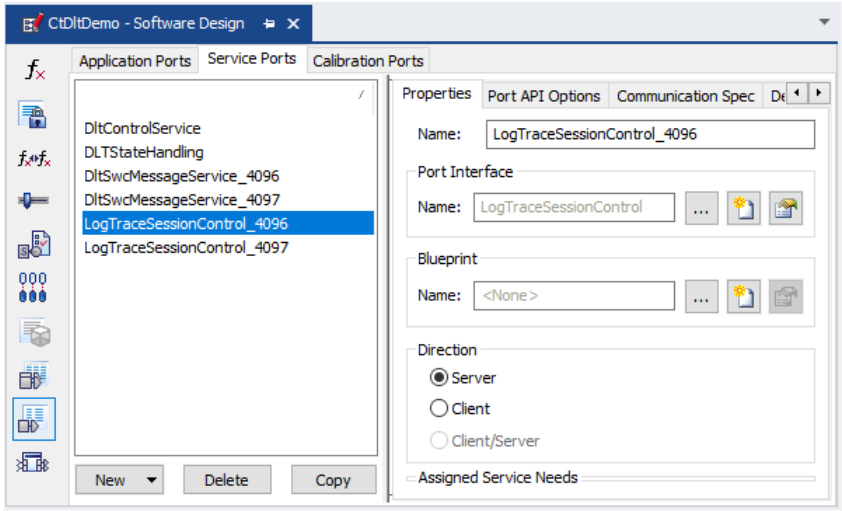
The DLT provides the possibility to inform the SWC that the threshold or the trace status of a DLT user (tuple of application id and context id) has changed. Therefore, the DLT provides the service port ‘LogTraceSessionControl’.

Add another service port from port interface (please refer to 3.5.1.2.2). Filter by ‘LogTrace’ and select the port interface ‘LogTraceSessionControl’ and click ‘OK’.

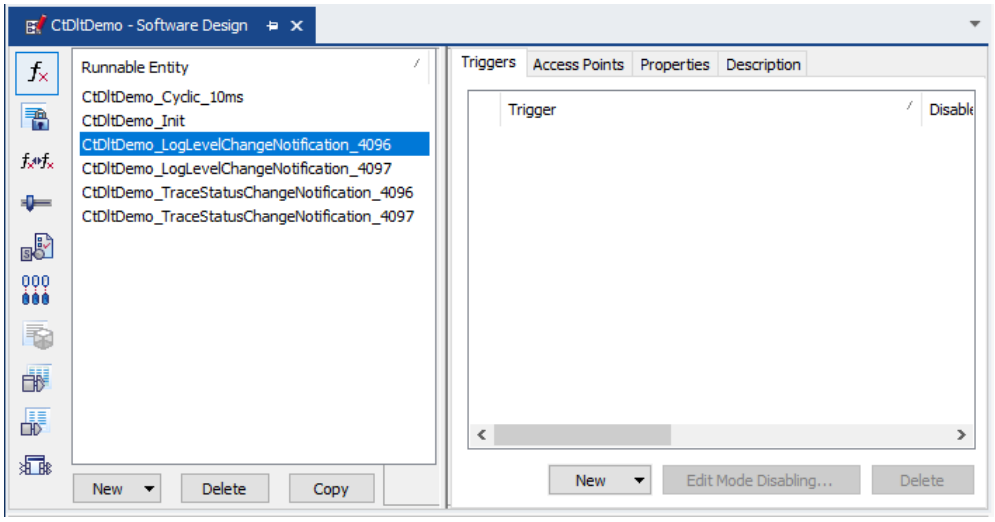


In this example, we create two `/MICROSAR/Dlt/DltSwc` which are both implemented by the SWC ‘CtDltDemo’, therefore we add another ‘LogTraceSessionControl’ and rename both like ‘LogTraceSessionControl_4096’ and ‘LogTraceSessionControl_4097’.

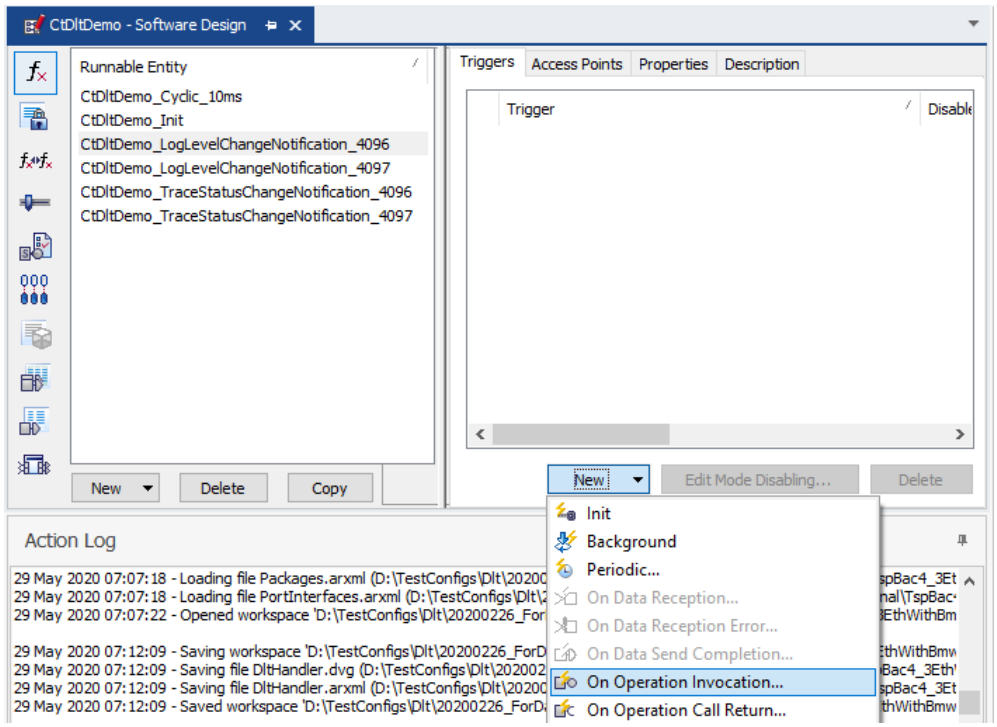
Set the Direction of both ports to ‘Server’.



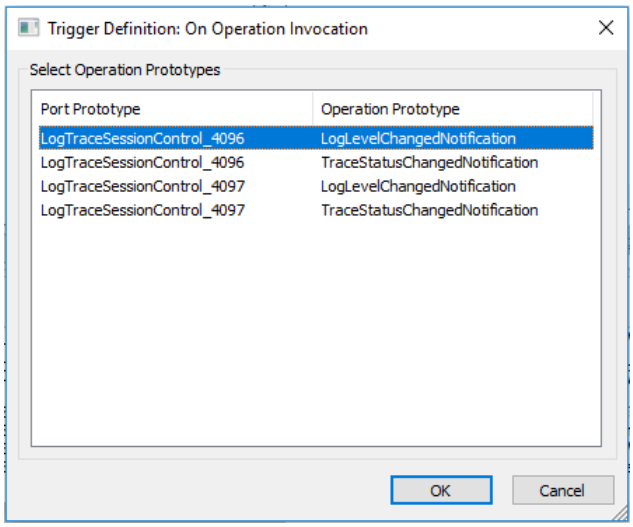
Because the SWC is notified by DLT if a threshold or trace status is change, we need for each ‘LogTraceSessionControl’ port and for each operation a new runnable. Please refer to section 3.5.1.2.3 to see how to add a runnable.



Now the triggers must be assigned. Therefore, select the runnable, click ‘Triggers’, ‘New’, and ‘On Operation Invokation…’.



Select the corresponding operation and repeat this for all ‘LogTraceSessionControl’ runnables.



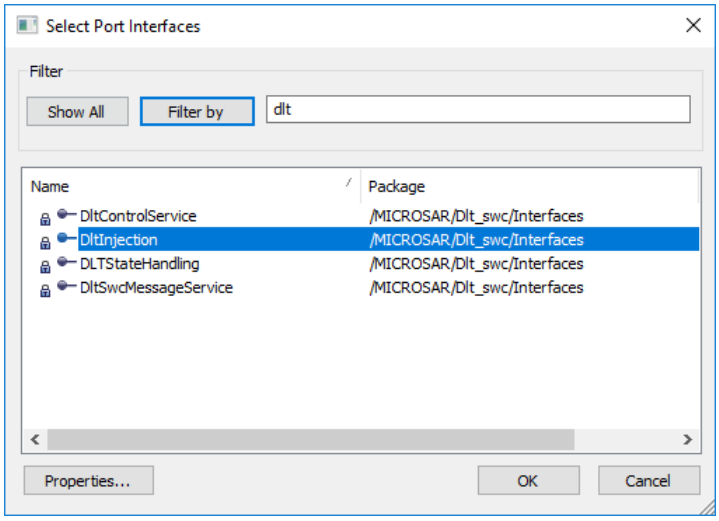
Now the SWC is always informed by DLT if a threshold or trace status of a DLT user is changed.

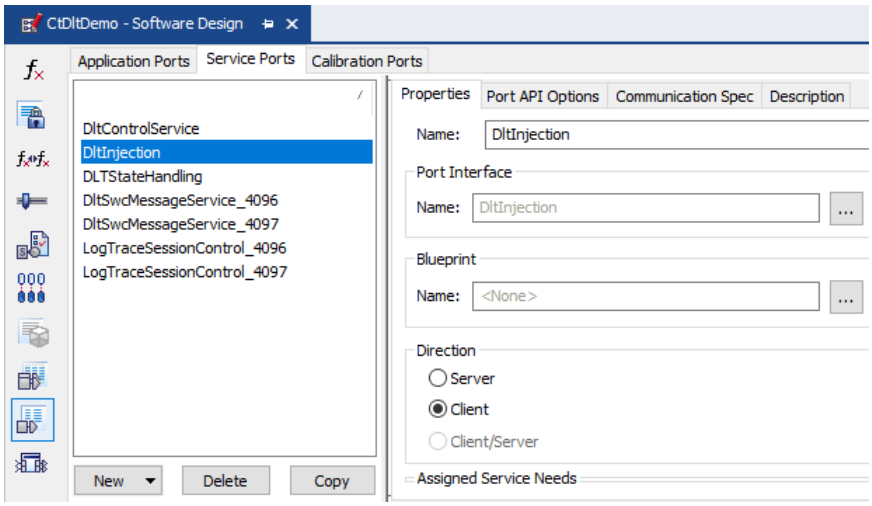
3.5.1.2.6 Injection request

The DLT provides the possibility that the SWC injects an application specific service in the same or another SWC. Therefore, the DLT provides the service port ‘DltInjection’.

Note that it makes more sense to inject a service in another SWC, but to demonstrate the functionality, in this example both is added to the same SWC ‘CtDltDemo’.

Add another service port from port interface (please refer to 3.5.1.2.2). Filter by ‘Dlt’ and select the port interface ‘DltInjection’ and click ‘OK’.

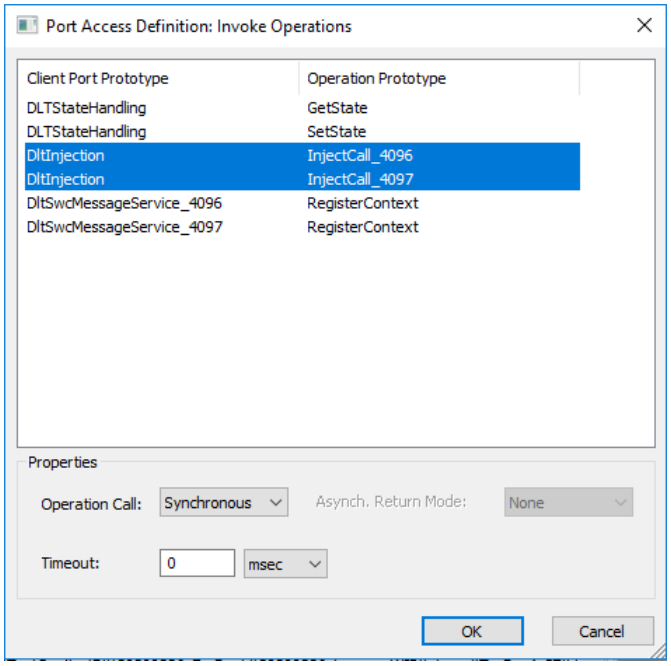




In this example, we extend the existing runnable ‘CtDltDemo_Cyclic_10ms’ by the operations of ‘DltControlService’. Therefore, we add more invoked operations (for more details please refer to section 3.5.1.2.3).

Select all required operations (in this example, all operations are added) and click ‘OK’.

Note that for each `/MICROSAR/Dlt/DltSwc` there is one explicit operation with the session id as postfix. Therefore, if there is no configured `DltSwc`, the injection feature is not available.

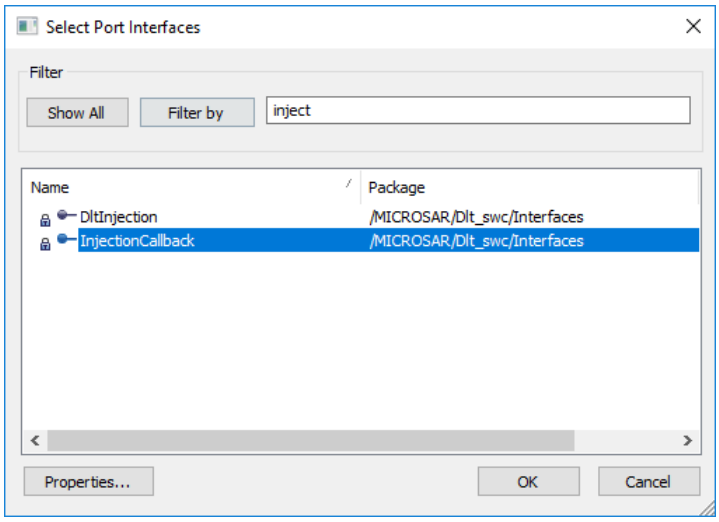


Now it is possible to inject service in other (or the same) SWC.

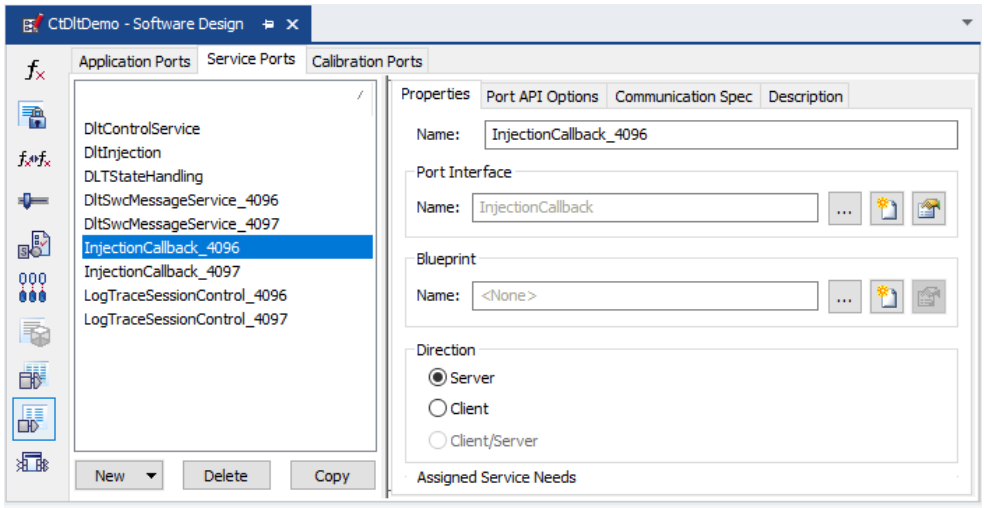
3.5.1.2.7 Injection callback

The DLT provides the possibility to inject application specific services in SWCs. Therefore, the DLT provides the service port ‘InjectionCallback’.

Add another service port from port interface (please refer to 3.5.1.2.2). Filter by ‘Injection’ and select the port interface ‘InjectionCallback’ and click ‘OK’.

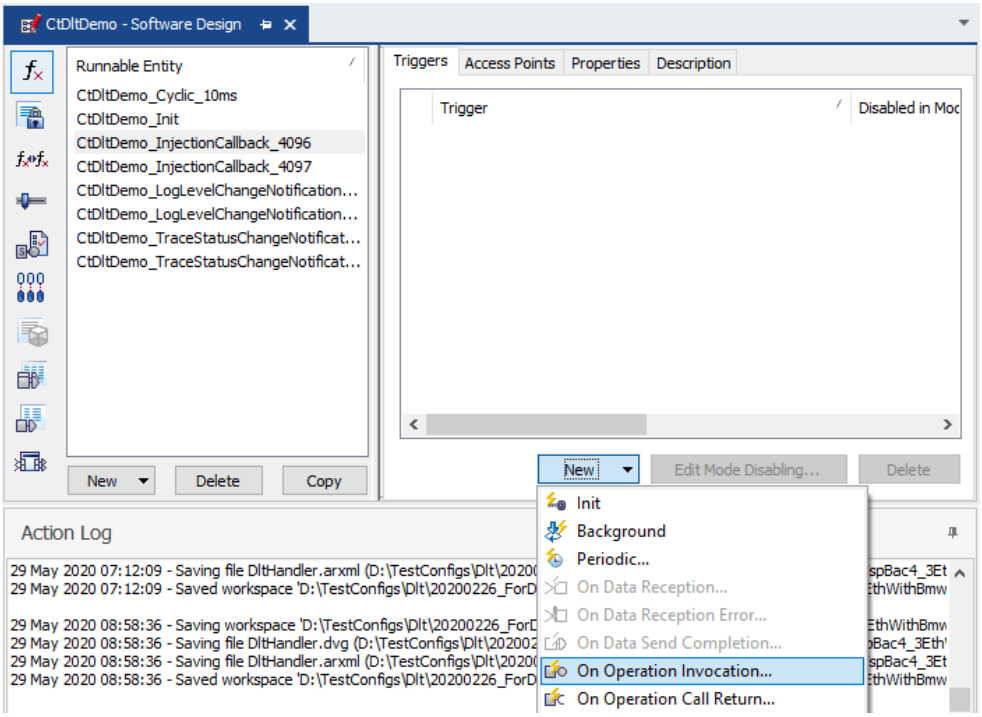


Because the SWC is injected by DLT for both configured `/MICROSAR/Dlt/DltSwc`, we need two ‘InjectionCallback’ ports. So, repeat the step above and rename the ports like ‘InjectionCallback_4096’ and ‘InjectionCallback_4097’. And set the ‘Direction’ of both ports to ‘Server’.

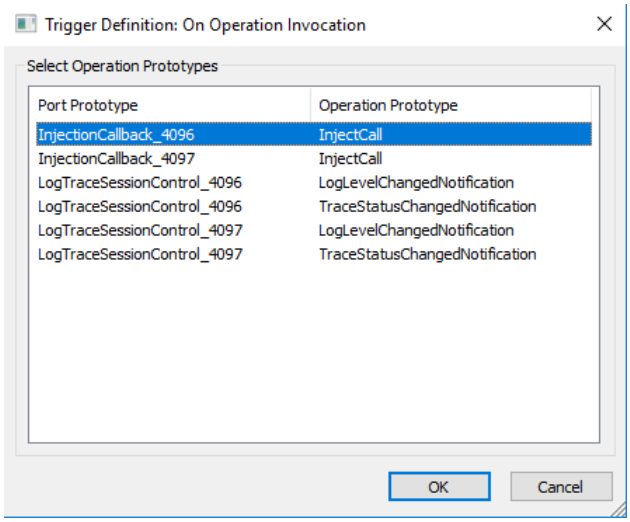


For each ‘InjectionCallback’ port a new runnable is required. Please refer to section 3.5.1.2.3 to see how to add a runnable.

Add new triggers to the new runnables, therefore select the runnable, click ‘Triggers’, ‘New’, and ‘On Operation Invocation…’.



Select the corresponding operation prototype and repeat this for the other injection callback runnable.

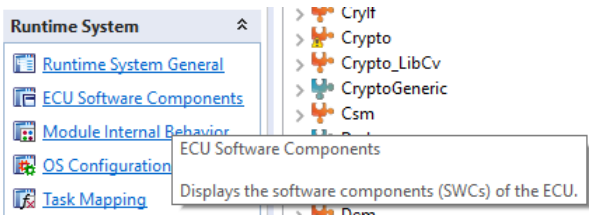


Now the SWC can receive service injections.

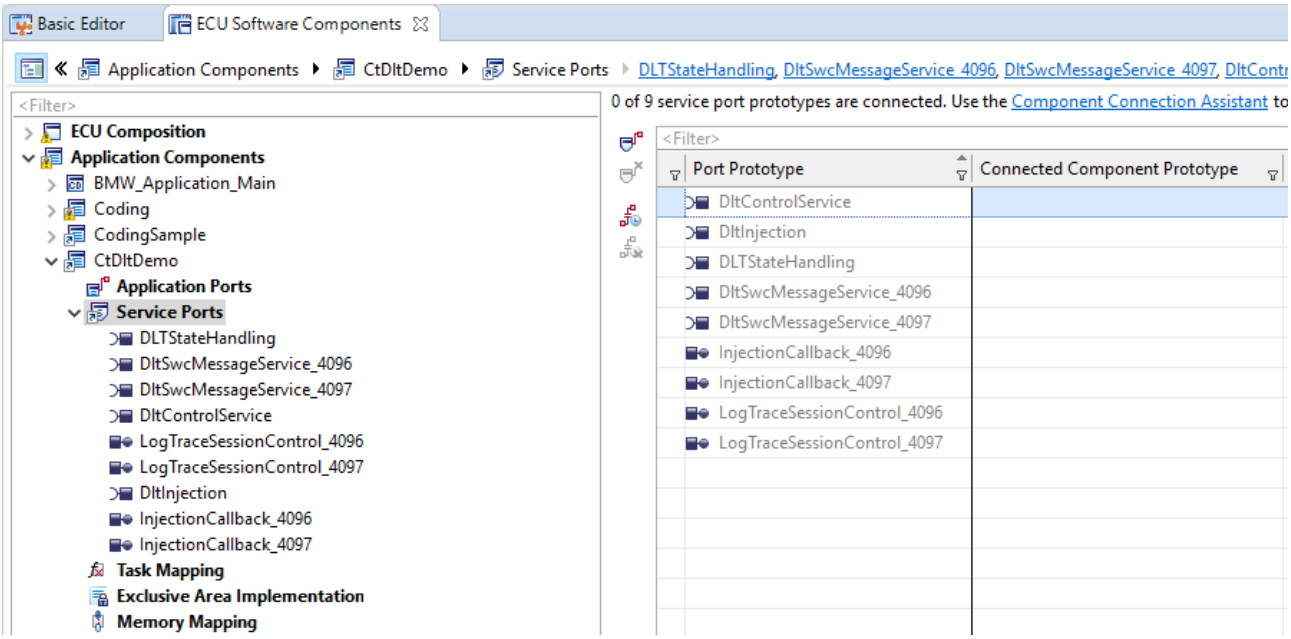
3.5.1.2.8 Update in DaVinci Configurator Pro

Now the ports and runnables are available, but they are not attached to the DLT.

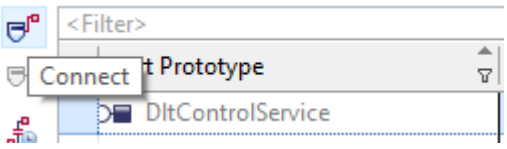
To do this now, open the comfort editor ‘ECU Software Components’.



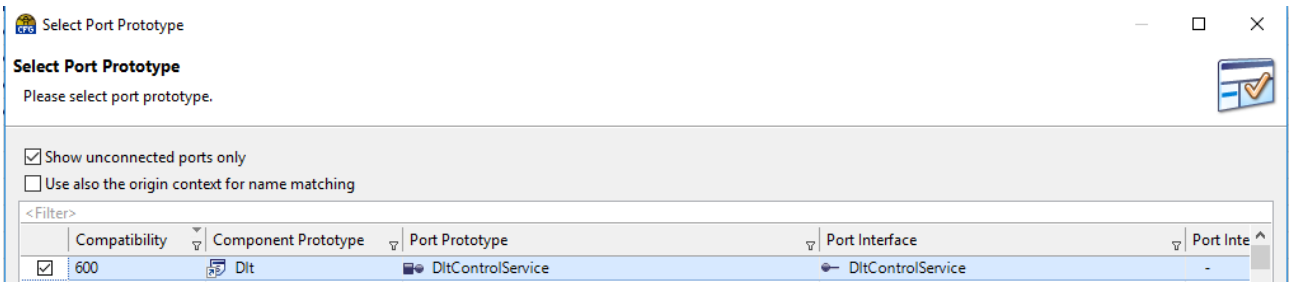
Select ‘Application Components’, ‘CtDltDemo’, and ‘Service Ports’. All service ports configured in the DaVinci Developer are displayed here.



Now, map each port to the corresponding DLT port by selecting the port and clicking ‘Connect’.

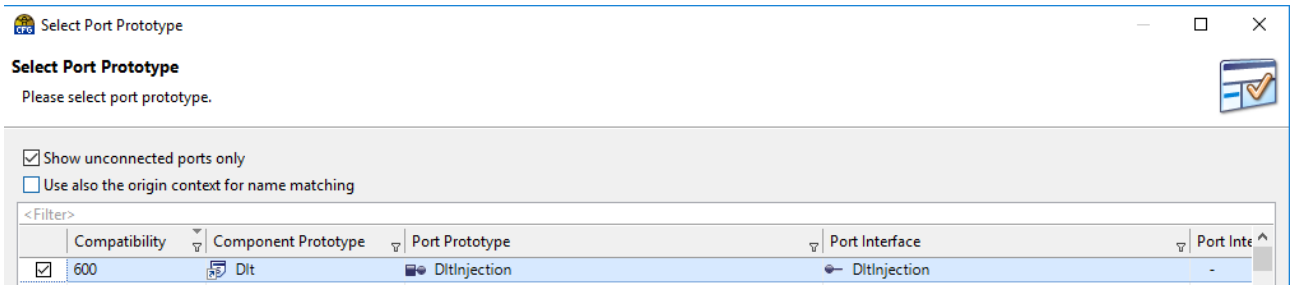


Select ‘Show unconnected ports only’, the upmost match should be already the correct one. Check this row and click ‘OK’.



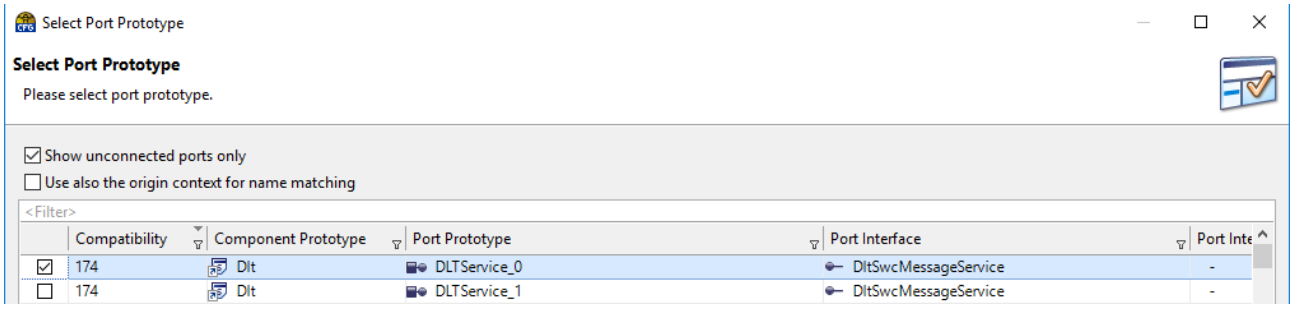
Repeat this for all other ports.

- DltInjection:

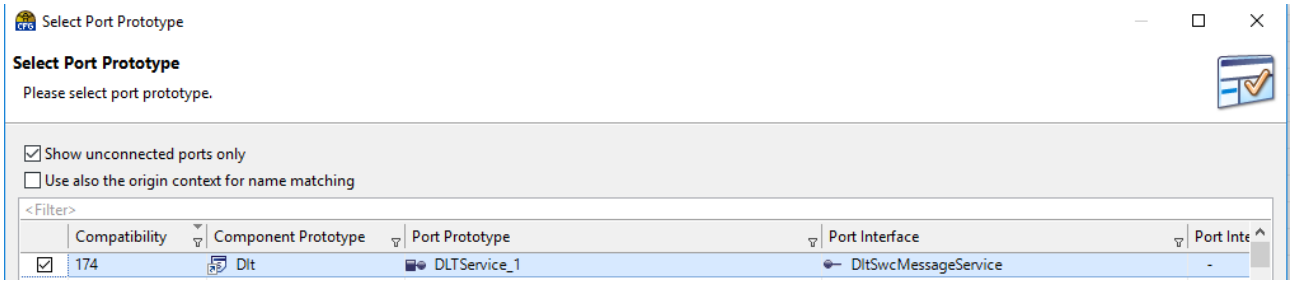


- DLTStateHandling:

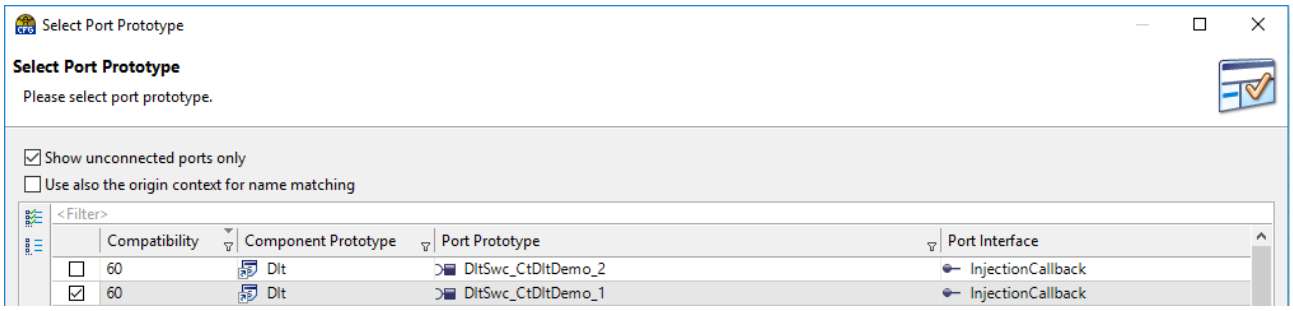
- DltSwcMessageService_4096:



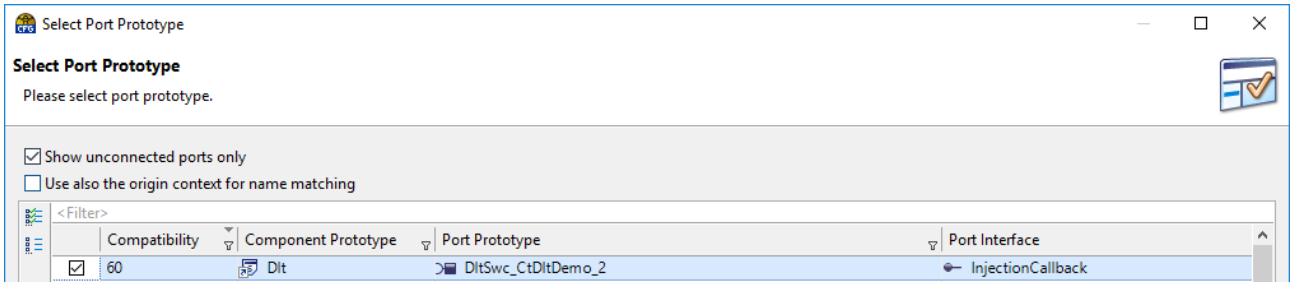
- DltSwcMessageService_4097:



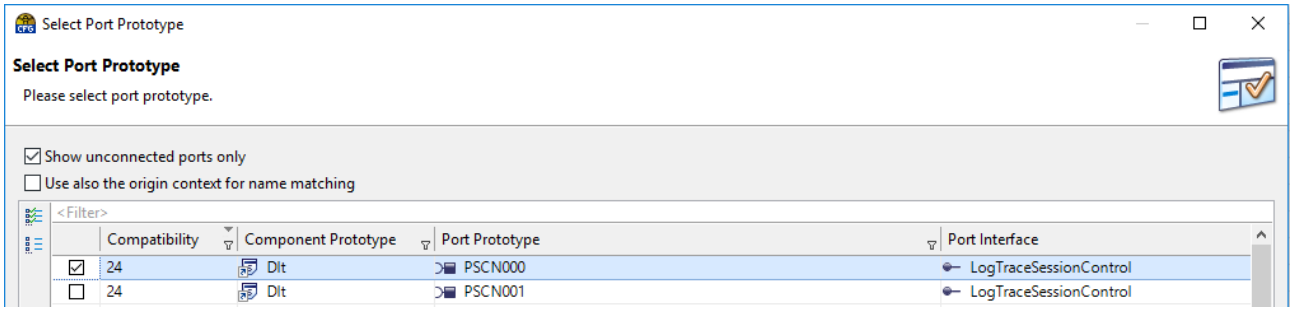
- InjectCallback_4096:
  - Note that DltSwc_CtDltDemo_1 has the session id 4096



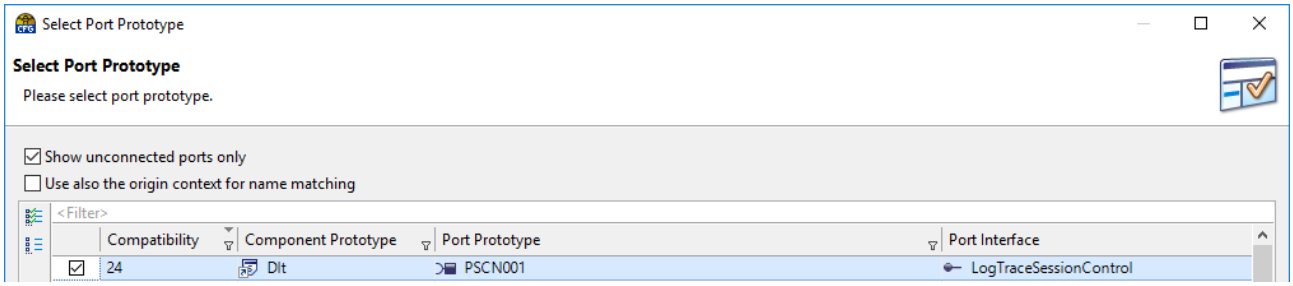
- InjectCallback_4097:
  - Note that DltSwc_CtDltDemo_2 has the session id 4097



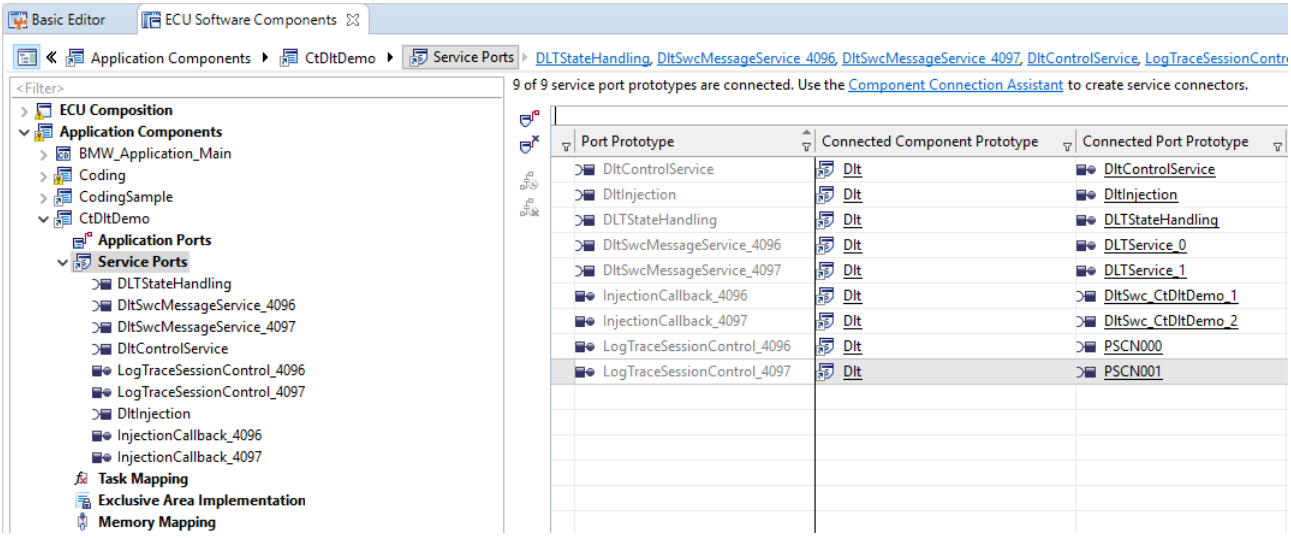
- LogTraceSessionControl_4096:



- LogTraceSessionControl_4097:

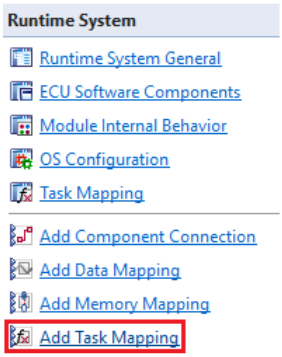


Now all ports are connected:

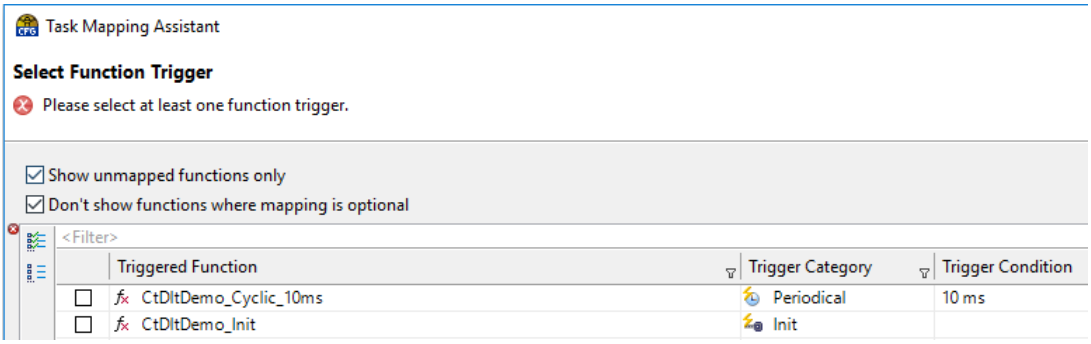


Task mapping:

Add task mapping for CtDltDemo_Init and CtDltDemo_Cyclic_10ms by clicking the comfort editor ‘Add Task Mapping’.

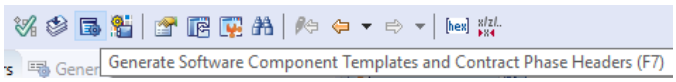


Select the unmapped service.



Then select the required task and change (if required) the order and click ‘Finish’. Repeat it for the other service.

Now the complete configuration should be done and working. Safe the changes and generate. Additionally, generate the SWC templates.



3.5.2 Information about DLTs' IDs

DLT requires a lot of different IDs. Thus, here is some information about these IDs:

The couple of application ID and context ID define one message of one DLT user. Multiple messages can be part of one session. Thus, in one session each couple of application ID and context ID must be unique. Because one application ID can contain multiple context IDs, each context ID must be unique in the scope of one application ID.

DLT supports two transmission modes:

- Verbose mode
- Non-verbose mode

In case of non-verbose mode, less data is transmitted to the DLT master (i.e. the 'static data' which is always constant is left away). For information about the DLT master, refer to chapter 3.5.7). This ('static') data is required for the DLT master to know. Thus, the DLT master requires another source for it. This source is a FIBEX file. To refer to the FIBEX-file message IDs are used. These message IDs must be transmitted within a non-verbose message and must be described in the FIBEX-file. Each DLT user requires one message ID.

Because multiple ECUs in one automobile should be able to use the DLT simultaneously, there is ECU IDs. These ECU IDs are unique in one automobile.

ID	relation	ID
Application ID	1:n	Context ID
Session ID	1:n	Application ID | Context ID
Message ID	1:1	Application ID | Context ID
DLT user	1:1	Session ID
ECU ID	1:n	Session ID

Table 3-5 Relations of IDs

3.5.3 DLT protocol

The DLT protocol is implemented as specified in chapter 7.7 of AUTOSAR DLT specification [1].

Length (bytes)	Name	Description
4,8,12 or 16	Standard Header (mandatory)	Contains essential information for interpreting the DLT message.
10	Extended Header (optional)	Can be added optionally for providing more information or for use in control messages.
x	Payload (optional)	Contains information about a specific log and trace message.

Table 3-6 DLT protocol as specified in [1]

In the standard header it is defined which information the header contains and how long the message is.

Caution
The bit MSBF (most significant bit first) in standard header only defines the endianness of payload. The headers (standard and extended) are always in big endian as specified by [PRS_Dlt_00091] in [6].

3.5.3.1 Verbose mode

If the SWC wants to send a log or trace message, it can be sent in verbose or in non-verbose mode.

If set to verbose mode, the message should send all dynamic and static data otherwise the static data should be replaced by the corresponding message ID. The message ID must be sent at first place in the payload.

In verbose mode the payload contains one or more couples of static and dynamic data. The static data is always 32bit and describes the following dynamic value. For more information about static and dynamic data in log and trace messages refer to [1].

3.5.4 Message types

There are three kinds of messages.

3.5.4.1 Log messages

Log messages are always sent from application via DLT to a DLT master.

The size of a log message can be between 8 and `DltLogChannelMaxMessageLength` bytes. It always contains the standard header. In non-verbose mode the 4-byte message ID is added. In verbose mode the extended header is added. Additionally, the optional payload is added.

3.5.4.2 Trace messages

Trace messages are always sent from application or RTE via DLT to a DLT master.

The size of a trace message can be between 8 and `DltLogChannelMaxMessageLength` bytes. It always contains the standard header. In non-verbose mode the 4-byte message ID is added. In verbose mode the extended header is added. Additionally, the optional payload is added.

3.5.4.3 Control messages

Control messages are always sent between DLT and DLT master. They are used to configure the registered contexts of DLT users. Furthermore, they are used by DLT master to request context information.

Control messages always contain a standard header, an extended header and a service ID. Additional payload is optional and depends on the service. Therefore, refer to chapter 5.3 of AUTOSAR DLT specification [6].

If the requested service is not supported, it returns a control message with one parameter matching “not supported”.

3.5.5 Message Filtering

To reduce the number of DLT messages, it is possible to filter messages which are currently not important.

The filtering can be done by DLT or already by the SWC (DLT user). The filtering in DLT is initially enabled and can be disabled by the API Dlt_SetMessageFiltering or a DLT service ‘SetMessageFilterering’ at runtime.

The SWCs can check the importance of an DLT message and decide what to do; reject or send it. Each DLT user has a log level threshold and a trace status, which are applied for filtering.

To enable an SWC to filter its DLT messages, the SWC must be configured and the following parameter must be checked:

> `/MICROSAR/Dlt/DltSwc/DltSwcSupportLogLevelChangeNotification`

If enabled, the SWC is notified about changed log level and trace status for its DLT users.

Filtering in the DLT has the advantage that the filtering is done at a central point instead of in each SWC.

The advantage of filtering in the SWCs is, that the runtime effort is reduced because the DLT messages can be rejected before the application assembled it.

3.5.6 Sending log and trace messages from SWC

If you have done all steps of configuration described in chapter 3.5.1, you can add the embedded part of communication between DLT and SWCs. Here the same example is used.

In Table 3-7 all information of the example is listed. There is one SWC (CtDltDemo), two runnables each handling 2 messages with different log levels and verbose modes.

Runnable	Message ID	Application ID	Context ID	Session ID	Log Level
CtDltDemo_ReceiveCanSignal	3	“CtD1” = 0x43744431	“ReC1” = 0x52654331	4096	Warn
CtDltDemo_ReceiveCanSignal	4	“CtD1” = 0x43744431	“ReC2” = 0x52654332	4096	Fatal
CtDltDemo_ReceiveFrSignal	5	“CtD2” = 0x43744432	“ReF1” = 0x52654631	4097	Error
CtDltDemo_ReceiveFrSignal	6	“CtD2” = 0x43744432	“ReF2” = 0x52654632	4097	Verb

Table 3-7 Example configuration of all DLT user messages

To reach this configuration some adaptions must be done in DaVinci Configurator Pro and DaVinci Developer:

Configuration parameter	state
`/MICROSAR/Dlt/DltGeneral/DltMaxNumberOfApplications`	> 4
`/MICROSAR/Dlt/DltGeneral/DltMaxNumberOfContextsPerApplication`	> 2
`/MICROSAR/Dlt/DltConfigSet/DltProtocol/DltUseVerboseMode`	Active
`/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltDefaultLogLevel`	DLT_LOG_INFO

/MICROSAR/Dlt/DltNonVerboseMessage /DltNonVerboseMessage_CanRx_1 -> Message Id = 3 /DltNonVerboseMessage_CanRx_2 -> Message Id = 4 /DltNonVerboseMessage_FrRx_1 -> Message Id = 5 /DltNonVerboseMessage_FrRx_2 -> Message Id = 6	

Table 3-8 Example configuration in DaVinci Configurator Pro

It is possible to register the users during runtime by calling Dlt_RegisterContext for each tuple of application id and context id. Please refer to section 3.5.6.1 for more information.

Alternatively, this can be done via configuration; please refer to the following table.

Configuration parameter	state
/MICROSAR/Dlt/DltSwc	DltSwc_CtDltDemo_1
/MICROSAR/Dlt/DltSwc/DltSwcSessionId	4096
/MICROSAR/Dlt/DltSwc/DltSwcContext	ReceiveCanSignal_1
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcApplicationId	CtD1
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcContextId	ReC1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold	Threshold_ReC1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltLogLevelThresholdSwcContextRef	ReceiveCanSignal_1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltThreshold	DLT_LOG_WARN
/MICROSAR/Dlt/DltSwc	DltSwc_CtDltDemo_1
/MICROSAR/Dlt/DltSwc/DltSwcSessionId	4096
/MICROSAR/Dlt/DltSwc/DltSwcContext	ReceiveCanSignal_2
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcApplicationId	CtD1
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcContextId	ReC2
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold	Threshold_ReC2
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltLogLevelThresholdSwcContextRef	ReceiveCanSignal_2
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltThreshold	DLT_LOG_FATAL
/MICROSAR/Dlt/DltSwc	DltSwc_CtDltDemo_2
/MICROSAR/Dlt/DltSwc/DltSwcSessionId	4097
/MICROSAR/Dlt/DltSwc/DltSwcContext	ReceiveFrSignal_1
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcApplicationId	CtD2
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcContextId	ReF1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold	Threshold_ReF1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltLogLevelThresholdSwcContextRef	ReceiveFrSignal_1
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltThreshold	DLT_LOG_ERROR
/MICROSAR/Dlt/DltSwc	DltSwc_CtDltDemo_2
/MICROSAR/Dlt/DltSwc/DltSwcSessionId	4097
/MICROSAR/Dlt/DltSwc/DltSwcContext	ReceiveFrSignal_2
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcApplicationId	CtD2
/MICROSAR/Dlt/DltSwc/DltSwcContext/DltSwcContextId	ReF2
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold	Threshold_ReF2

/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltLogLevelThresholdSwcContextRef	ReceiveFrSignal_2
/MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltLogLevelThreshold/DltThreshold	DLT_LOG_VERBOSE

Table 3-9 Optional configuration of SWCs and their contexts

In DaVinci Developer the runnable “CtDltDemo_ReceiveCanSignal” should access the port DltSwcMessageService_4096.SendLogMessage. The runnable “CtDltDemo_ReceiveFrSignal” should access DltSwcMessageService_4097.SendLogMessage.

Generate the SWCs and the DLT.

3.5.6.1 Context Registration

If the registration is not done via configuration (please refer to Table 3-9), the SWC has to register its context/s before it can use the DLT services. In the example before, we added an init-runnable to the SWC and named it CtDltDemo_Init. Within this runnable the contexts can be registered by calling following functions:

- Rte_Call_DltSwcMessageService_4096_RegisterContext
- Rte_Call_DltSwcMessageService_4097_RegisterContext
- …

Each API should be called for each message of one session ID.

You must pass an application ID and a context ID (all other parameter can be 0 respectively NULL_PTR). For the example, call both APIs twice to register all four couples of application and context IDs, according Table 3-7.

A possible implementation is shown in the following example.

Example

```
#include "Rte_CtDltDemo.h"
#include "Dlt.h"
…

Dlt_AppIDsType CtDltDemo_DltContext[2]; /* Array size == number of used session IDs */
…

FUNC(void, CtDltDemo_CODE) CtDltDemo_Init(void)
{
  Dlt_ReturnType retVal;

  CtDltDemo_DltContext[0].app_id = 0x43744431; /* CtD1 */
  CtDltDemo_DltContext[0].app_description = NULL_PTR;
  CtDltDemo_DltContext[0].len_app_description = 0;
  CtDltDemo_DltContext[0].count_context_ids = 2;

  CtDltDemo_DltContext[0].context_id_info[0].context_id = 0x52654331; /* ReC1 */
  CtDltDemo_DltContext[0].context_id_info[0].log_level = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[0].context_id_info[0].trace_status = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[0].context_id_info[0].len_context_description = 0;
  CtDltDemo_DltContext[0].context_id_info[0].context_description = NULL_PTR;
  CtDltDemo_DltContext[0].context_id_info[0].cbk_info.verbose_mode = 0; /* Will be set
by DLT */

  CtDltDemo_DltContext[0].context_id_info[1].context_id = 0x52654332; /* ReC2 */
  CtDltDemo_DltContext[0].context_id_info[1].log_level = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[0].context_id_info[1].trace_status = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[0].context_id_info[1].len_context_description = 0;
  CtDltDemo_DltContext[0].context_id_info[1].context_description = NULL_PTR;

  CtDltDemo_DltContext[1].app_id = 0x43744432; /* CtD1 */
  CtDltDemo_DltContext[1].app_description = NULL_PTR;
  CtDltDemo_DltContext[1].len_app_description = 0;
  CtDltDemo_DltContext[1].count_context_ids = 2;

  CtDltDemo_DltContext[1].context_id_info[0].context_id = 0x52654631; /* ReF1 */
  CtDltDemo_DltContext[1].context_id_info[0].log_level = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[1].context_id_info[0].trace_status = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[1].context_id_info[0].len_context_description = 0;
  CtDltDemo_DltContext[1].context_id_info[0].context_description = NULL_PTR;
  CtDltDemo_DltContext[1].context_id_info[0].cbk_info.verbose_mode = 0; /* Will be set
by DLT */

  CtDltDemo_DltContext[1].context_id_info[1].context_id = 0x52654632; /* ReF2 */
  CtDltDemo_DltContext[1].context_id_info[1].log_level = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[1].context_id_info[1].trace_status = 0; /* Will be set by DLT */
  CtDltDemo_DltContext[1].context_id_info[1].len_context_description = 0;
  CtDltDemo_DltContext[1].context_id_info[1].context_description = NULL_PTR;

  retVal = (Dlt_ReturnType)Rte_Call_DltSwcMessageService_4096_RegisterContext(
                            CtDltDemo_DltContext[0].app_id,
                            CtDltDemo_DltContext[0].context_id_info[0].context_id,
                            NULL_PTR, 0, NULL_PTR, 0);

  /* Check return value */

  retVal = (Dlt_ReturnType)Rte_Call_DltSwcMessageService_4096_RegisterContext(
                            CtDltDemo_DltContext[0].app_id,
                            CtDltDemo_DltContext[0].context_id_info[1].context_id,
                            NULL PTR, 0, NULL PTR, 0);

  /* Check return value */

  retVal = (Dlt_ReturnType)Rte_Call_DltSwcMessageService_4097_RegisterContext(
                            CtDltDemo_DltContext[1].app_id,
                            CtDltDemo DltContext[1].context id info[0].context id,
                            NULL_PTR, 0, NULL_PTR, 0);

  /* Check return value */

  retVal = (Dlt_ReturnType)Rte_Call_DltSwcMessageService_4097_RegisterContext(
                            CtDltDemo_DltContext[1].app_id,
```

```
                                CtDltDemo_DltContext[1].context_id_info[1].context_id,
                                NULL_PTR, 0, NULL_PTR, 0);
  /* Check return value */
}
```

Note
Only the contexts of VFB traces and SWCs must be registered. The contexts of DEM and DET events are initially registered and can be used from ECU start up on.

For each succeeded registration DLT calls the following APIs of SWC (here: CtDltDemo):

- CtDltDemo_LogLevelChangedNotification_<XYZ>
- CtDltDemo_TraceStatusChangedNotification_<XYZ>

Where <XYZ> stands for the session ID. With a call to these APIs the SWC is informed about the log level, trace status and the verbose mode it should be using. The APIs can also be called after initialization, triggered by the DLT master. All APIs get an application ID and a context ID, with these IDs the SWC knows which message context must be changed.

Example

```
FUNC(Std_ReturnType, CtDltDemo_CODE) CtDltDemo_SetLogLevel_4096(Dlt_ApplicationIDType
        AppId, Dlt_ContextIDType ContextId, Dlt_MessageLogLevelType LogLevel)
{
  Std_ReturnType retVal = E_NOT_OK;

  if (CtDltDemo_DltContext[0].app_id == AppId)
  {
    if (CtDltDemo_DltContext[0].context_id_info[0].context_id == ContextId)
    {
      CtDltDemo_DltContext[0].context_id_info[0].log_level = (sint8)LogLevel;
      retVal = E_OK;
    }
    else if (CtDltDemo_DltContext[0].context_id_info[1].context_id == ContextId)
    {
      CtDltDemo_DltContext[0].context_id_info[1].log_level = (sint8)LogLevel;
      retVal = E_OK;
    }
  }

  return retVal;
}
/* Do the same thing for CtDltDemo_SetTraceStatus_4096/4097,
CtDltDemo_SetVerboseMode_4096/4097 and CtDltDemo_SetLogLevel_4097 */
```

Now all messages have the default value for log level, trace status and verbose mode. Thus, the values depend on your configuration in DaVinci Configurator Pro:

- /MICROSAR/Dlt/DltConfigSet/DltLogLevelSetting/DltDefaultLogLevel
- /MICROSAR/Dlt/DltConfigSet/DltTraceStatusSetting/DltDefaultTraceStatus

To change these values to the required, the DLT master (e.g. DltViewer) must trigger the services:

- Set_LogLevel
- Set_TraceStatus

Or the application must call the following APIs:

- Dlt_SetLogLevel
- Dlt_SetTraceStatus

Therefore, each couple of Application ID and Context ID must be set individually or globally. In the example in Table 3-7, each value must be change individually because they differ in value.

3.5.6.2 Sending of log and trace messages

As soon as the context is registered, log and trace messages can be sent. Therefore, the function like macros can be called in context of the configured runnables:

- Rte_Call_DltSwcMessageService_4096_SendLogMessage
- Rte_Call_DltSwcMessageService_4097_SendLogMessage
- Rte_Call_DltSwcMessageService_4096_SendTraceMessage
- Rte_Call_DltSwcMessageService_4097_SendTraceMessage
- …

The log message service requires log info, log data and log data length. The parameters are explained in chapter 4.2.7, but session ID is not part of the signature. The session ID is set by the RTE.

The trace message service requires trace info, trace data and trace data length. The parameters are explained in chapter 4.2.8, but session ID is not part of the signature. The session ID is set by the RTE.

Example

```
FUNC(void, CtDltDemo_CODE) CtDltDemo_ReceiveCanSignal(void)
{
  Dlt_ReturnType retVal;
  Dlt_MessageLogInfoType logInfo; /* arg1 */
  uint8 logData[DLT_MAX_MESSAGE_LENGTH];
  uint16 logDataIndex = 0;

  /* Optional: Only required if you want to check if the message is allowed to be send
               in verbose mode. */
  boolean useExtendedHeader = FALSE;
  boolean useVerboseMode    = FALSE;

  /* Dynamic data for payload */
  uint16 logVar1;
  uint8  logVar2;

  /*
   *  Send first log message in non-verbose mode
   */
  logInfo.arg_count =  2; /* There are 2 log variables (logVar1 and logVar2) */
  logInfo.app_id =     CtDltDemo_DltContext[0].app_id;
  logInfo.context_id = CtDltDemo_DltContext[0].context_id_info[0].context_id;
  logInfo.options =    DLT_NON_VERBOSE_MSG | DLT_TYPE_LOG;
  logDataIndex =       7; /* Message ID (4Byte) + logVar1 (2Byte) + logvar2 (1Byte) */
  logVar1 =            0xFFFF;
  logVar2 =            0x01;

  /* Optional: Check if log level in pass through range. Check can be omitted if
   /MICROSAR/Dlt/DltMultipleConfigurationContainer/DltMessageFiltering/DltFilterMessages
     is active.
   */
  if (CtDltDemo_DltContext[0].context_id_info[0].log_level <=
      (sint8)Dlt_DefaultMaxLogLevel)
  {
    /* Check endianness */
#if defined (DLT_HEADER_PAYLOAD_BYTEORDER) && (DLT_HEADER_PAYLOAD_BYTEORDER ==
DLT_BIGENDIAN)
    /* Set message ID as first parameter in payload */
    logData[0] =
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >> 24);
    logData[1] =
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >> 16);
    logData[2] =
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >>  8);
    logData[3] = (uint8)(DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1);
    logData[4] = (uint8)(logVar1 >> 8); /* First log variable */
    logData[5] = (uint8)(logVar1);      /* First log variable */
    logData[6] = (uint8)(logVar2); /* Second log variable */
#else
    /* Set message ID as first parameter in payload */
    logData[0] = (uint8)(DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1);
    logData[1] =
```

```
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >>  8);
    logData[2] =
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >> 16);
    logData[3] =
(uint8)((uint32)DltConf_DltNonVerboseMessage_DltNonVerboseMessage_CanRx_1 >> 24);
    logData[4] = (uint8)(logVar1);      /* First log variable */
    logData[5] = (uint8)(logVar1 >> 8); /* First log variable */
    logData[6] = (uint8)(logVar2); /* Second log variable */
#endif

    retVal = Rte_Call_DltSwcMessageService_4096_SendLogMessage( &logInfo, logData,
logDataIndex);
    /* Check return value… */
  }

  /*
   *  Send second log message in verbose mode
   */
  logInfo.arg_count =  2; /* There are 2 log variables (logVar1 and logVar2) */
  logInfo.app_id =     CtDltDemo_DltContext[0].app_id;
  logInfo.context_id = CtDltDemo_DltContext[0].context_id_info[1].context_id;
  logInfo.options = DLT_VERBOSE_MSG | DLT_TYPE_LOG;
  logDataIndex = 11; /* Type Info (4Byte) + logVar1 (2Byte) + Type info (4Byte) +
                        logvar2 (1Byte) */
  logVar1 = 0x0001;
  logVar2 = 0xFF;

  /* Optional: Check if verbose mode active */
#if defined (DLT_IMPLEMENT_EXTENDED_HEADER)
  useExtendedHeader = Dlt_HeaderUseExtendedHeader;
#endif /* (DLT_IMPLEMENT_EXTENDED_HEADER) */
#if defined (DLT_IMPLEMENT_VERBOSE_MODE)
  useVerboseMode = Dlt_HeaderUseVerboseMode;
#endif /* (DLT_IMPLEMENT_VERBOSE_MODE) */

  if ((useVerboseMode == TRUE) && (useExtendedHeader == TRUE))
  {
    /* Optional check if log level in pass through range. Can be omitted if
   /MICROSAR/Dlt/DltMultipleConfigurationContainer/DltMessageFiltering/DltFilterMessages
       is active.
     */
    if (CtDltDemo_DltContext[0].context_id_info[1].log_level <=
        (sint8)Dlt_DefaultMaxLogLevel)
    {
      /* Check endianness */
#if defined (DLT_HEADER_PAYLOAD_BYTEORDER) && (DLT_HEADER_PAYLOAD_BYTEORDER ==
DLT_BIGENDIAN)
      logData[ 0] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 1] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 2] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 3] = 0x42; /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 4] = (uint8)(logVar1 >> 8); /* First log variable */
      logData[ 5] = (uint8)(logVar1);      /* First log variable */
      logData[ 6] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
```

```
      logData[ 7] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[ 8] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[ 9] = 0x41; /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[10] = (uint8)(EventStatusNew); /* EventStatusNew */
#else
      logData[ 0] = 0x42; /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 1] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 2] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 3] = 0;    /* Type Info - logVar1 - Unsigned int, 16 bit */
      logData[ 4] = (uint8)(logVar1);      /* First log variable */
      logData[ 5] = (uint8)(logVar1 >> 8); /* First log variable */
      logData[ 6] = 0x41; /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[ 7] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[ 8] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[ 9] = 0;    /* Type Info - logVar2 - Unsigned int, 8 bit */
      logData[10] = (uint8)(logVar2); /* Second log variable */
#endif

      /* Send log message */
      retVal = Rte_Call_DltSwcMessageService_4096_SendLogMessage(&logInfo, logData,
                                                                 logDataIndex);
      /* Check return value… */
    }
  }
}

/* Do the same for CtDltDemo_ReceiveFrSignal… */
```

3.5.7 DLT master

The DLT requires a master to be triggered and controlled. CANoe and CANape do not have a native support of DLT protocol (used by Dlt on PduR). Therefore, another tool could be used. For example, the DltViewer of GENIVI (http://projects.genivi.org/diagnostic-log-trace/home) can be used.

3.5.8 FIBEX File

The DLT master requires a FIBEX file to interpret the non-verbose messages of DLT correctly. In the payload of a non-verbose message only dynamic data (and the message ID) is set. Thus, in the FIBEX file the corresponding static data, to interpret these dynamic data, is provided.

The FIBEX file is composed of five sections which are explained in Table 3-10. It contains the required content for DEM and DET events and the content of the examples of Table 3-7.

Section	Description

`<?xml version="1.0" encoding="UTF-8" standalone="no"?>`
`<fx:FIBEX xmlns:can="http://www.asam.net/xml/fbx/can"`
`xmlns:fx="http://www.asam.net/xml/fbx"`
`xmlns:ho="http://www.asam.net/xml" VERSION="3.1.0"`
`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
`xsi:schemaLocation="http://www.asam.net/xml/fbx`
`xml_schema\fibex.xsd http://www.asam.net/xml/fbx/can`
	Define FIBEX version, project ID and name.

Section	Description
xml_schema\fibex4can.xsd"> <fx:PROJECT ID="DiagnosticLogAndTrace"> <ho:SHORT-NAME>DLT</ho:SHORT-NAME> </fx:PROJECT> <fx:ELEMENTS>	
<!-- ECU --> <fx:ECUS> <fx:ECU ID="ECUI"> <ho:SHORT-NAME>ECU ID</ho:SHORT-NAME> <fx:MANUFACTURER-EXTENSION> <SW_VERSION>001.000.000</SW_VERSION> <APPLICATIONS> <APPLICATION> <APPLICATION_ID>DEM</APPLICATION_ID> <APPLICATION_DESCRIPTION>DEM event</APPLICATION_DESCRIPTION> <CONTEXTS> <CONTEXT> <CONTEXT_ID>STD0</CONTEXT_ID> <CONTEXT_DESCRIPTION>Dlt_DemTriggerOnEventStatus</CONTEXT_DESCRIPTION> </CONTEXT> </CONTEXTS> </APPLICATION> <APPLICATION> <APPLICATION_ID>DET</APPLICATION_ID> <APPLICATION_DESCRIPTION>DET event</APPLICATION_DESCRIPTION> <CONTEXTS> <CONTEXT> <CONTEXT_ID>STD</CONTEXT_ID> <CONTEXT_DESCRIPTION>Dlt_DetForwardErrorTrace</CONTEXT_DESCRIPTION> </CONTEXT> </CONTEXTS> </APPLICATION> <APPLICATION> <APPLICATION_ID>CtD1</APPLICATION_ID> <APPLICATION_DESCRIPTION>SWC CtDltDemo</APPLICATION_DESCRIPTION> <CONTEXTS> <CONTEXT> <CONTEXT_ID>ReC1</CONTEXT_ID> <CONTEXT_DESCRIPTION>ReceiveCanSignal 1</CONTEXT_DESCRIPTION> </CONTEXT> <CONTEXT> <CONTEXT_ID>ReC2</CONTEXT_ID> <CONTEXT_DESCRIPTION>ReceiveCanSignal 2</CONTEXT_DESCRIPTION> </CONTEXT> </CONTEXTS> </APPLICATION> <APPLICATION> <APPLICATION_ID>CtD2</APPLICATION_ID> <APPLICATION_DESCRIPTION>SWC CtDltDemo</APPLICATION_DESCRIPTION> <CONTEXTS> <CONTEXT> <CONTEXT_ID>ReF1</CONTEXT_ID> <CONTEXT_DESCRIPTION>ReceiveFrSignal 1</CONTEXT_DESCRIPTION> </CONTEXT> <CONTEXT> <CONTEXT_ID>ReF2</CONTEXT_ID> <CONTEXT_DESCRIPTION>ReceiveFrSignal 2</CONTEXT_DESCRIPTION> </CONTEXT> </CONTEXTS> </APPLICATION> </APPLICATIONS>	Define the ECU ID, the Software version, all application IDs and their corresponding context IDs. In the example on the left, only application ID and the corresponding context ID of DEM events, DET events and the examples of Table 3-7 are defined. More application and context IDs can be added.

Section	Description
</fx:MANUFACTURER-EXTENSION> </fx:ECU> </fx:ECUS>	
<!-- PDUS --> <fx:PDUS> <!--================ Message 1 ==================--> <!--================ 1. Parameter ==================--> <fx:PDU ID="PDU_1_0"> <ho:SHORT-NAME>PDU_1_0</ho:SHORT-NAME> <ho:DESC>Event ID:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!--================ 2. Parameter ==================--> <fx:PDU ID="PDU_1_1"> <ho:SHORT-NAME>PDU_1_1</ho:SHORT-NAME> <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_1_0"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT16"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> <!--================ 3. Parameter ==================--> <fx:PDU ID="PDU_1_2"> <ho:SHORT-NAME>PDU_1_2</ho:SHORT-NAME> <ho:DESC>New Event Status:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!--================ 4. Parameter ==================--> <fx:PDU ID="PDU_1_3"> <ho:SHORT-NAME>PDU_1_3</ho:SHORT-NAME> <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_1_1"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT8"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> <!--================ Message 2 ==================--> <!--================ 1. Parameter ==================--> <fx:PDU ID="PDU_2_0"> <ho:SHORT-NAME>PDU_2_0</ho:SHORT-NAME> <ho:DESC>Module ID:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!--================ 2. Parameter ==================--> <fx:PDU ID="PDU_2_1"> <ho:SHORT-NAME>PDU_2_1</ho:SHORT-NAME> <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_2_0"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT16"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> <!--================ 3. Parameter ==================--> <fx:PDU ID="PDU_2_2"> <ho:SHORT-NAME>PDU_2_2</ho:SHORT-NAME> <ho:DESC>Instance ID:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!--================ 4. Parameter ==================--> <fx:PDU ID="PDU_2_3">	Define all PDUs (== Message ID) and their parameter. In the example on the left only DEM events are defined (message ID == 1). The DEM event sends 2 parameter values to DLT master, they are called “S_1_0” and “S_1_0”. To add more information in the message, both parameter values get another parameter with a describing string. Thus, “1. Parameter” provides the string “Event ID:”. “2. Parameter” provides the first received parameter with type uint16. “3. Parameter” provides the string “New Event Status:” and “4. Parameter” provides the corresponding parameter value from type uint8. Messages for DET and the examples of Table 3-7 are also described. More messages and/or parameter can be added if required.

Section	Description

```
      <ho:SHORT-NAME>PDU_2_3</ho:SHORT-NAME>
      <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
      <fx:SIGNAL-INSTANCES>
        <fx:SIGNAL-INSTANCE ID="S_2_1">
          <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          <fx:SIGNAL-REF ID-REF="S_UINT8"/>
        </fx:SIGNAL-INSTANCE>
      </fx:SIGNAL-INSTANCES>
    </fx:PDU>
    <!--================ 5. Parameter ==================-->
    <fx:PDU ID="PDU_2_4">
      <ho:SHORT-NAME>PDU_2_4</ho:SHORT-NAME>
      <ho:DESC>API ID:</ho:DESC>
      <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    </fx:PDU>
    <!--================ 6. Parameter ==================-->
    <fx:PDU ID="PDU_2_5">
      <ho:SHORT-NAME>PDU_2_5</ho:SHORT-NAME>
      <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
      <fx:SIGNAL-INSTANCES>
        <fx:SIGNAL-INSTANCE ID="S_2_2">
          <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          <fx:SIGNAL-REF ID-REF="S_UINT8"/>
        </fx:SIGNAL-INSTANCE>
      </fx:SIGNAL-INSTANCES>
    </fx:PDU>
    <!--================ 7. Parameter ==================-->
    <fx:PDU ID="PDU_2_6">
      <ho:SHORT-NAME>PDU_2_6</ho:SHORT-NAME>
      <ho:DESC>Error ID:</ho:DESC>
      <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    </fx:PDU>
    <!--================ 8. Parameter ==================-->
    <fx:PDU ID="PDU_2_7">
      <ho:SHORT-NAME>PDU_2_7</ho:SHORT-NAME>
      <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
      <fx:SIGNAL-INSTANCES>
        <fx:SIGNAL-INSTANCE ID="S_2_3">
          <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          <fx:SIGNAL-REF ID-REF="S_UINT8"/>
        </fx:SIGNAL-INSTANCE>
      </fx:SIGNAL-INSTANCES>
    </fx:PDU>
    <!--================ Message 3 ==================-->
    <!--================ 1. Parameter ==================-->
    <fx:PDU ID="PDU_3_0">
      <ho:SHORT-NAME>PDU_3_0</ho:SHORT-NAME>
      <ho:DESC>Log variable 1:</ho:DESC>
      <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    </fx:PDU>
    <!--================ 2. Parameter ==================-->
    <fx:PDU ID="PDU_3_1">
      <ho:SHORT-NAME>PDU_3_1</ho:SHORT-NAME>
      <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>
      <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
      <fx:SIGNAL-INSTANCES>
        <fx:SIGNAL-INSTANCE ID="S_3_0">
          <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          <fx:SIGNAL-REF ID-REF="S_UINT16"/>
        </fx:SIGNAL-INSTANCE>
      </fx:SIGNAL-INSTANCES>
    </fx:PDU>
    <!--================ 3. Parameter ==================-->
    <fx:PDU ID="PDU_3_2">
      <ho:SHORT-NAME>PDU_3_2</ho:SHORT-NAME>
      <ho:DESC>Log variable 2:</ho:DESC>
      <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
```

Section	Description

```
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
  </fx:PDU>
  <!--================ 4. Parameter ==================-->
  <fx:PDU ID="PDU_3_3">
    <ho:SHORT-NAME>PDU_3_3</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    <fx:SIGNAL-INSTANCES>
      <fx:SIGNAL-INSTANCE ID="S_3_1">
        <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
        <fx:SIGNAL-REF ID-REF="S_UINT8"/>
      </fx:SIGNAL-INSTANCE>
    </fx:SIGNAL-INSTANCES>
  </fx:PDU>
  <!--================ Message 4 ==================-->
  <!--================ 1. Parameter ==================-->
  <fx:PDU ID="PDU_4_0">
    <ho:SHORT-NAME>PDU_4_0</ho:SHORT-NAME>
    <ho:DESC>Log variable 1:</ho:DESC>
    <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
  </fx:PDU>
  <!--================ 2. Parameter ==================-->
  <fx:PDU ID="PDU_4_1">
    <ho:SHORT-NAME>PDU_4_1</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    <fx:SIGNAL-INSTANCES>
      <fx:SIGNAL-INSTANCE ID="S_4_0">
        <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
        <fx:SIGNAL-REF ID-REF="S_UINT16"/>
      </fx:SIGNAL-INSTANCE>
    </fx:SIGNAL-INSTANCES>
  </fx:PDU>
  <!--================ 3. Parameter ==================-->
  <fx:PDU ID="PDU_4_2">
    <ho:SHORT-NAME>PDU_4_2</ho:SHORT-NAME>
    <ho:DESC>Log variable 2:</ho:DESC>
    <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
  </fx:PDU>
  <!--================ 4. Parameter ==================-->
  <fx:PDU ID="PDU_4_3">
    <ho:SHORT-NAME>PDU_4_3</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    <fx:SIGNAL-INSTANCES>
      <fx:SIGNAL-INSTANCE ID="S_4_1">
        <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
        <fx:SIGNAL-REF ID-REF="S_UINT8"/>
      </fx:SIGNAL-INSTANCE>
    </fx:SIGNAL-INSTANCES>
  </fx:PDU>
  <!--================ Message 5 ==================-->
  <!--================ 1. Parameter ==================-->
  <fx:PDU ID="PDU_5_0">
    <ho:SHORT-NAME>PDU_5_0</ho:SHORT-NAME>
    <ho:DESC>Log variable 1:</ho:DESC>
    <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
  </fx:PDU>
  <!--================ 2. Parameter ==================-->
  <fx:PDU ID="PDU_5_1">
    <ho:SHORT-NAME>PDU_5_1</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>OTHER</fx:PDU-TYPE>
    <fx:SIGNAL-INSTANCES>
      <fx:SIGNAL-INSTANCE ID="S_5_0">
        <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
        <fx:SIGNAL-REF ID-REF="S_UINT16"/>
      </fx:SIGNAL-INSTANCE>
    </fx:SIGNAL-INSTANCES>
  </fx:PDU>
```

Section	Description
<!--================ 3. Parameter ==================--> <fx:PDU ID="PDU_5_2"> <ho:SHORT-NAME>PDU_5_2</ho:SHORT-NAME> <ho:DESC>Log variable 2:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!--================ 4. Parameter ==================--> <fx:PDU ID="PDU_5_3"> <ho:SHORT-NAME>PDU_5_3</ho:SHORT-NAME> <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_5_1"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT8"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> </fx:PDUS>	
<!-- FRAMES --> <fx:FRAMES> <!-- 1. Log and Trace Message --> <fx:FRAME ID="ID_1"> <ho:SHORT-NAME>ID_1</ho:SHORT-NAME> <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH> <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE> <fx:PDU-INSTANCES> <fx:PDU-INSTANCE ID="P_1_0"> <fx:PDU-REF ID-REF="PDU_1_0"/> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_1_1"> <fx:PDU-REF ID-REF="PDU_1_1"/> <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_1_2"> <fx:PDU-REF ID-REF="PDU_1_2"/> <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_1_3"> <fx:PDU-REF ID-REF="PDU_1_3"/> <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> </fx:PDU-INSTANCES> <fx:MANUFACTURER-EXTENSION> <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE> <MESSAGE_INFO>DLT_LOG_INFO</MESSAGE_INFO> <APPLICATION_ID>DEM</APPLICATION_ID> <CONTEXT_ID>STD0</CONTEXT_ID> <MESSAGE_SOURCE_FILE>Dlt.c</MESSAGE_SOURCE_FILE> <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER> </fx:MANUFACTURER-EXTENSION> </fx:FRAME> <!-- 2. Log and Trace Message --> <fx:FRAME ID="ID_2"> <ho:SHORT-NAME>ID_2</ho:SHORT-NAME> <fx:BYTE-LENGTH>5</fx:BYTE-LENGTH> <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE> <fx:PDU-INSTANCES> <fx:PDU-INSTANCE ID="P_2_0"> <fx:PDU-REF ID-REF="PDU_2_0"/> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_2_1"> <fx:PDU-REF ID-REF="PDU_2_1"/> <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_2_2"> <fx:PDU-REF ID-REF="PDU_2_2"/> <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_2_3">	Define all messages with all parameter (also the describing parameters), the message type, the message info, the application ID, the context ID, message source file and the message line number. In the example on the left only the DEM event message is described. All parameter defined in the PDUS-section are listed. The parameter “P_1_1” has type uint16 and parameter “P_1_3” has type uint8, thus the byte length has to be set to 3. The message info can be changed at runtime, thus it is possible that the log level within the ECU differs to the log level in the FIBEX file.

Section	Description

```
            <fx:PDU-REF ID-REF="PDU_2_3"/>
            <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_2_4">
            <fx:PDU-REF ID-REF="PDU_2_4"/>
            <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_2_5">
            <fx:PDU-REF ID-REF="PDU_2_5"/>
            <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_2_6">
            <fx:PDU-REF ID-REF="PDU_2_6"/>
            <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_2_7">
            <fx:PDU-REF ID-REF="PDU_2_7"/>
            <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
        </fx:PDU-INSTANCES>
        <fx:MANUFACTURER-EXTENSION>
          <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE>
          <MESSAGE_INFO>DLT_LOG_INFO</MESSAGE_INFO>
          <APPLICATION_ID>DET</APPLICATION_ID>
          <CONTEXT_ID>STD</CONTEXT_ID>
          <MESSAGE_SOURCE_FILE>Dlt.c</MESSAGE_SOURCE_FILE>
          <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER>
        </fx:MANUFACTURER-EXTENSION>
      </fx:FRAME>
      <!-- 3. Log and Trace Message -->
      <fx:FRAME ID="ID_3">
        <ho:SHORT-NAME>ID_3</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH>
        <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE>
        <fx:PDU-INSTANCES>
          <fx:PDU-INSTANCE ID="P_3_0">
            <fx:PDU-REF ID-REF="PDU_3_0"/>
            <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_3_1">
            <fx:PDU-REF ID-REF="PDU_3_1"/>
            <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_3_2">
            <fx:PDU-REF ID-REF="PDU_3_2"/>
            <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_3_3">
            <fx:PDU-REF ID-REF="PDU_3_3"/>
            <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
        </fx:PDU-INSTANCES>
        <fx:MANUFACTURER-EXTENSION>
          <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE>
          <MESSAGE_INFO>DLT_LOG_WARN</MESSAGE_INFO>
          <APPLICATION_ID>CtD1</APPLICATION_ID>
          <CONTEXT_ID>ReC1</CONTEXT_ID>

<MESSAGE_SOURCE_FILE>CtDltDemo.c</MESSAGE_SOURCE_FILE>
          <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER>
        </fx:MANUFACTURER-EXTENSION>
      </fx:FRAME>
      <!-- 4. Log and Trace Message -->
      <fx:FRAME ID="ID_4">
        <ho:SHORT-NAME>ID_4</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH>
        <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE>
        <fx:PDU-INSTANCES>
          <fx:PDU-INSTANCE ID="P_4_0">
            <fx:PDU-REF ID-REF="PDU_4_0"/>
            <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_4_1">
```

Section	Description

```
            <fx:PDU-REF ID-REF="PDU_4_1"/>
            <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_4_2">
            <fx:PDU-REF ID-REF="PDU_4_2"/>
            <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_4_3">
            <fx:PDU-REF ID-REF="PDU_4_3"/>
            <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
        </fx:PDU-INSTANCES>
        <fx:MANUFACTURER-EXTENSION>
          <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE>
          <MESSAGE_INFO>DLT_LOG_FATAL</MESSAGE_INFO>
          <APPLICATION_ID>CtD1</APPLICATION_ID>
          <CONTEXT_ID>ReC2</CONTEXT_ID>

<MESSAGE_SOURCE_FILE>CtDltDemo.c</MESSAGE_SOURCE_FILE>
          <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER>
        </fx:MANUFACTURER-EXTENSION>
      </fx:FRAME>
      <!-- 5. Log and Trace Message -->
      <fx:FRAME ID="ID_5">
        <ho:SHORT-NAME>ID_5</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH>
        <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE>
        <fx:PDU-INSTANCES>
          <fx:PDU-INSTANCE ID="P_5_0">
            <fx:PDU-REF ID-REF="PDU_5_0"/>
            <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_5_1">
            <fx:PDU-REF ID-REF="PDU_5_1"/>
            <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_5_2">
            <fx:PDU-REF ID-REF="PDU_5_2"/>
            <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_5_3">
            <fx:PDU-REF ID-REF="PDU_5_3"/>
            <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
        </fx:PDU-INSTANCES>
        <fx:MANUFACTURER-EXTENSION>
          <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE>
          <MESSAGE_INFO>DLT_LOG_ERROR</MESSAGE_INFO>
          <APPLICATION_ID>CtD2</APPLICATION_ID>
          <CONTEXT_ID>ReF1</CONTEXT_ID>

<MESSAGE_SOURCE_FILE>CtDltDemo.c</MESSAGE_SOURCE_FILE>
          <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER>
        </fx:MANUFACTURER-EXTENSION>
      </fx:FRAME>
     <!-- 6. Log and Trace Message -->
      <fx:FRAME ID="ID_6">
        <ho:SHORT-NAME>ID_6</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH>
        <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE>
        <fx:PDU-INSTANCES>
          <fx:PDU-INSTANCE ID="P_6_0">
            <fx:PDU-REF ID-REF="PDU_6_0"/>
            <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_6_1">
            <fx:PDU-REF ID-REF="PDU_6_1"/>
            <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
          <fx:PDU-INSTANCE ID="P_6_2">
            <fx:PDU-REF ID-REF="PDU_6_2"/>
            <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER>
          </fx:PDU-INSTANCE>
```

Section	Description
<fx:PDU-INSTANCE ID="P_6_3"> <fx:PDU-REF ID-REF="PDU_6_3"/> <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> </fx:PDU-INSTANCES> <fx:MANUFACTURER-EXTENSION> <MESSAGE_TYPE>DLT_TYPE_LOG</MESSAGE_TYPE> <MESSAGE_INFO>DLT_LOG_VERB</MESSAGE_INFO> <APPLICATION_ID>CtD2</APPLICATION_ID> <CONTEXT_ID>ReF2</CONTEXT_ID> <MESSAGE_SOURCE_FILE>CtDltDemo.c</MESSAGE_SOURCE_FILE> <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER> </fx:MANUFACTURER-EXTENSION> </fx:FRAME> </fx:FRAMES>	
<!-- signals --> <fx:SIGNALS> <!-- BOOL --> <fx:SIGNAL ID="S_BOOL"> <ho:SHORT-NAME>S_BOOL</ho:SHORT-NAME> <fx:CODING-REF ID-REF="BOOL"/> </fx:SIGNAL> <!-- SINT8 --> <fx:SIGNAL ID="S_SINT8"> <ho:SHORT-NAME>S_SINT8</ho:SHORT-NAME> <fx:CODING-REF ID-REF="SINT8"/> </fx:SIGNAL> <!-- UINT8 --> <fx:SIGNAL ID="S_UINT8"> <ho:SHORT-NAME>S_UINT8</ho:SHORT-NAME> <fx:CODING-REF ID-REF="UINT8"/> </fx:SIGNAL> <!-- SINT16 --> <fx:SIGNAL ID="S_SINT16"> <ho:SHORT-NAME>S_SINT16</ho:SHORT-NAME> <fx:CODING-REF ID-REF="SINT16"/> </fx:SIGNAL> <!-- UINT16 --> <fx:SIGNAL ID="S_UINT16"> <ho:SHORT-NAME>S_UINT16</ho:SHORT-NAME> <fx:CODING-REF ID-REF="UINT16"/> </fx:SIGNAL> <!-- SINT32 --> <fx:SIGNAL ID="S_SINT32"> <ho:SHORT-NAME>S_SINT32</ho:SHORT-NAME> <fx:CODING-REF ID-REF="SINT32"/> </fx:SIGNAL> <!-- UINT32 --> <fx:SIGNAL ID="S_UINT32"> <ho:SHORT-NAME>S_UINT32</ho:SHORT-NAME> <fx:CODING-REF ID-REF="UINT32"/> </fx:SIGNAL> <!-- SINT64 --> <fx:SIGNAL ID="S_SINT64"> <ho:SHORT-NAME>S_SINT64</ho:SHORT-NAME> <fx:CODING-REF ID-REF="SINT64"/> </fx:SIGNAL> <!-- UINT64 --> <fx:SIGNAL ID="S_UINT64"> <ho:SHORT-NAME>S_UINT64</ho:SHORT-NAME> <fx:CODING-REF ID-REF="UINT64"/> </fx:SIGNAL> <!-- FLOA16 --> <fx:SIGNAL ID="S_FLOA16"> <ho:SHORT-NAME>S_FLOA16</ho:SHORT-NAME> <fx:CODING-REF ID-REF="FLOA16"/> </fx:SIGNAL> <!-- FLOA32 --> <fx:SIGNAL ID="S_FLOA32"> <ho:SHORT-NAME>S_FLOA32</ho:SHORT-NAME> <fx:CODING-REF ID-REF="FLOA32"/> </fx:SIGNAL>	This field is constant and can be copied without adaption. It defines the signal types.

Section	Description
<!-- FLOA64 --> <fx:SIGNAL ID="S_FLOA64"> <ho:SHORT-NAME>S_FLOA64</ho:SHORT-NAME> <fx:CODING-REF ID-REF="FLOA64"/> </fx:SIGNAL> <!-- LENGTH --> <fx:SIGNAL ID="S_LENGTH"> <ho:SHORT-NAME>S_LENGTH</ho:SHORT-NAME> <fx:CODING-REF ID-REF="LENGTH"/> </fx:SIGNAL> <!-- STRG_ASCII --> <fx:SIGNAL ID="S_STRG_ASCII"> <ho:SHORT-NAME>S_STRG_ASCII</ho:SHORT-NAME> <fx:CODING-REF ID-REF="STRG_ASCII"/> </fx:SIGNAL> <!-- STRG_UTF8 --> <fx:SIGNAL ID="S_STRG_UTF8"> <ho:SHORT-NAME>S_STRG_UTF8</ho:SHORT-NAME> <fx:CODING-REF ID-REF="STRG_UTF8"/> </fx:SIGNAL> <!-- RAWD --> <fx:SIGNAL ID="S_RAWD"> <ho:SHORT-NAME>S_RAWD</ho:SHORT-NAME> <fx:CODING-REF ID-REF="RAWD"/> </fx:SIGNAL> </fx:SIGNALS> </fx:ELEMENTS>	
<!-- PROCESSING INFORMATION --> <fx:PROCESSING-INFORMATION> <!-- codings --> <fx:CODINGS> <fx:CODING ID="BOOL"> <ho:SHORT-NAME>BOOL</ho:SHORT-NAME> <ho:DESC>Coding for boolean values.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT8"> <ho:BIT-LENGTH>8</ho:BIT-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="SINT8"> <ho:SHORT-NAME>SINT8</ho:SHORT-NAME> <ho:DESC>Coding for signed 8bit values.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="SIGNED" ho:BASE-DATA-TYPE="A_INT8"> <ho:BIT-LENGTH>8</ho:BIT-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="UINT8"> <ho:SHORT-NAME>UINT8</ho:SHORT-NAME> <ho:DESC>Coding for unsigned 8bit values.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT8"> <ho:BIT-LENGTH>8</ho:BIT-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="SINT16"> <ho:SHORT-NAME>SINT16</ho:SHORT-NAME> <ho:DESC>Coding for signed 16bit values.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="SIGNED" ho:BASE-DATA-TYPE="A_INT16"> <ho:BIT-LENGTH>16</ho:BIT-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="UINT16"> <ho:SHORT-NAME>UINT16</ho:SHORT-NAME> <ho:DESC>Coding for unsigned 16bit values.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT16"> <ho:BIT-LENGTH>16</ho:BIT-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="SINT32"> <ho:SHORT-NAME>SINT32</ho:SHORT-NAME>	This field is constant and can be copied without adaption. It defines the types of the type info field ([Dlt135] of [1]).

Section	Description

```
        <ho:DESC>Coding for signed 32bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="SIGNED" ho:BASE-DATA-TYPE="A_INT32">
          <ho:BIT-LENGTH>32</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="UINT32">
        <ho:SHORT-NAME>UINT32</ho:SHORT-NAME>
        <ho:DESC>Coding for unsigned 32bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT32">
          <ho:BIT-LENGTH>32</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="SINT64">
        <ho:SHORT-NAME>SINT64</ho:SHORT-NAME>
        <ho:DESC>Coding for signed 64bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="SIGNED" ho:BASE-DATA-TYPE="A_INT64">
          <ho:BIT-LENGTH>64</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="UINT64">
        <ho:SHORT-NAME>UINT64</ho:SHORT-NAME>
        <ho:DESC>Coding for unsigned 64bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT64">
          <ho:BIT-LENGTH>64</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="FLOA16">
        <ho:SHORT-NAME>FLOA16</ho:SHORT-NAME>
        <ho:DESC>Coding for float 16bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="IEEE-FLOATING-TYPE" ho:BASE-DATA-TYPE="A_FLOAT32">
          <ho:BIT-LENGTH>16</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="FLOA32">
        <ho:SHORT-NAME>FLOA32</ho:SHORT-NAME>
        <ho:DESC>Coding for float 32bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="IEEE-FLOATING-TYPE" ho:BASE-DATA-TYPE="A_FLOAT32">
          <ho:BIT-LENGTH>32</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="FLOA64">
        <ho:SHORT-NAME>FLOA64</ho:SHORT-NAME>
        <ho:DESC>Coding for float 64bit values.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="IEEE-FLOATING-TYPE" ho:BASE-DATA-TYPE="A_FLOAT64">
          <ho:BIT-LENGTH>64</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="LENGTH">
        <ho:SHORT-NAME>LENGTH</ho:SHORT-NAME>
        <ho:DESC>Coding for length information.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_UINT16">
          <ho:BIT-LENGTH>16</ho:BIT-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="STRG_ASCII">
        <ho:SHORT-NAME>STRG_ASCII</ho:SHORT-NAME>
        <ho:DESC>Coding for ASCII string.</ho:DESC>
        <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE"
TERMINATION="ZERO" ho:BASE-DATA-TYPE="A_ASCIISTRING">
          <ho:MIN-LENGTH>0</ho:MIN-LENGTH>
          <ho:MAX-LENGTH>5120</ho:MAX-LENGTH>
        </ho:CODED-TYPE>
      </fx:CODING>
      <fx:CODING ID="STRG_UTF8">
        <ho:SHORT-NAME>STRG_UTF8</ho:SHORT-NAME>
```

Section	Description
<ho:DESC>Coding for UTF8 string.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="UTF-8" TERMINATION="ZERO" ho:BASE-DATA-TYPE="A_UNICODE2STRING"> <ho:MIN-LENGTH>0</ho:MIN-LENGTH> <ho:MAX-LENGTH>5120</ho:MAX-LENGTH> </ho:CODED-TYPE> </fx:CODING> <fx:CODING ID="RAWD"> <ho:SHORT-NAME>RAWD</ho:SHORT-NAME> <ho:DESC>Coding for raw data.</ho:DESC> <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE" ENCODING="UNSIGNED" ho:BASE-DATA-TYPE="A_BYTEFIELD"> <ho:MIN-LENGTH>0</ho:MIN-LENGTH> <ho:MAX-LENGTH>5120</ho:MAX-LENGTH> </ho:CODED-TYPE> </fx:CODING> </fx:CODINGS> </fx:PROCESSING-INFORMATION> </fx:FIBEX>	

Table 3-10 Structure of FIBEX file (complete FIBEX file with one example)

3.5.9 VFB tracing

VFB tracing is the implicit tracing of communication between SWCs via RTE. The SWCs do not send trace messages explicitly. Therefore, the RTE supports hook functions which are called before and after the execution of an RTE service. These hooks can be individually activated and deactivated within the DaVinci Configurator Pro by `/MICROSAR/Rte/RteGeneration/RteVfbTraceFunction`. To use them the global switch `/MICROSAR/Rte/RteGeneration/RteVfbTraceEnabled` must be set.

It is not required to configure the DaVinci Developer.

Note
It should be enough to use the start **or** the return hook for VFB tracing.

Caution
VFB tracing is not supported explicitly, but it is possible to implement it manually.

The chosen hook functions must be implemented manually, only the declaration is available in Rte_Hook.h.

3.5.9.1 Context Registration for VFB tracing

There are three different ways to register the contexts of VFB tracing.

3.5.9.1.1 Registration by configuration

This is the easiest way to register the DLT user for VFB tracing.

Add a DLT SWC:

> /MICROSAR/Dlt/DltSwc

Uncheck the following parameter:

> ./DltSwcSupportLogLevelChangeNotification

Add as many as required DLT SWC contexts:

> ./DltSwcContext

Set the application id of all DLT SWC contexts to “VFBT” (= 0x56464254).

> ./DltSwcApplicationId

Set the contexts to unique values:

> ./DltSwcContextId

Caution
For each added DLT SWC context, the parameter /MICROSAR/Dlt/DltGeneral/DltMaxNumberOfContextsPerApplication must be incremented.
Each increment increases the ROM and RAM usage of DLT.

3.5.9.1.2 Registration from hook

Precondition for this method is the following configuration:

> In DaVinci Configurator Pro for each VFB trace context one /MICROSAR/Dlt/DltNonVerboseMessage (= message ID) must be added. No more configuration must be done (do not increment the number of max DLT users).

> In DaVinci Developer no configuration must be done.

Within the hook function the DLT user context must be registered once. Therefore, call Dlt_RegisterContext directly. The first expected parameter is the session ID. For SWC this parameter is set by RTE, in case of VFB tracing the session ID must be set manually. Choose the session ID higher than 4096 and unequal to the already used session IDs.

Table 3-11 shows all parameter to be passed to Dlt_RegisterContext.

Parameter	Recommended Value	Description
Parg0 (session ID)	> 4096	Value must be greater than 4096 and unique in the ECU. One session ID for all VFB trace contexts is sufficient.
AppId	“VFBT” = 0x56464254	The application ID for VFB tracing has to be the four ASCII letters “VFBT”.
ContextId	wxyz	The context ID has to be chosen uniquely for the VFB tracing.
AppDescription	NULL_PTR	The application ID description is not

Parameter	Recommended Value	Description
		supported.
LenAppDescription	0	Must be 0 if AppDescription is NULL_PTR.
ContextDescription	NULL_PTR	The context ID description is not supported.
LenContextDescription	0	Must be 0 if ContextDescription is NULL_PTR.

Table 3-11 Recommended Parameter of Dlt_RegisterContext for VFB tracing

With this method the VFB tracing is clearly distinct from the logging and tracing of SWCs and it is easier to configure. But each time a hook using VFB tracing is called, the context must be checked if it is already registered. This increases the runtime of RTE.

3.5.9.1.3 Registration from SWC

To avoid the check if the context is already registered within the hook function, it is possible to register the VFB trace context within an init runnable of a SWC. Therefore, a separated DLT user (= Session ID) should be created, thus the used VFB trace hooks have their own session ID.

Required configuration in DaVinci Configurator Pro:

> add one /MICROSAR/Dlt/DltNonVerboseMessage (= message ID) per VFB trace context

> add one /MICROSAR/Dlt/DltSwc or use an existing SWC (= session ID; one is sufficient)

Required configuration in DaVinci Developer:

> Add an init runnable to the chosen SWC (if it does not already exist)

> Add a service port to the SWC and name it LogTraceSessionControl_<SessionID> (must have Server direction, the name may differ)

> For this session ID no service port with client direction has to be added (Dlt_SendTraceMessage is not send from SWC)

The parameters that have to be used for VFB trace context registration are described in Table 3-11.

This method increases configuration effort, but reduces implementation effort and runtime of RTE.

Note
The session ID, application ID and context ID of context registration has to be used for sending trace messages in the hook function.

3.5.9.2 Sending VFB trace messages

After the succeeded context registration, the function Dlt_SendTraceMessage can be called to send a VFB tracing message. This function has to be called within the hook function.

Usually all parameter of the hook function have to be concatenated in the trace data, but it is also possible to disclaim some or all parameter. At least the message ID (first parameter in trace data) has to be set.

In Table 3-12 shows all parameter to be passed to Dlt_SendTraceMessage.

Parameter	Recommended Value	Description
Parg0 (session ID)	> 4096	The registered session ID for this VFB trace context.
TraceInfo .app_id .conId .options .trace_info	 =“VFBT” =wxyz =DLT_NON_VERBOSE_MSG | DLT_TYPE_NW_TRACE =DLT_NW_TRACE_IPC	 Use always “VFBT” as application ID. Use the hook specific context ID. Set option to “non-verbose” and the message type to “network trace”. Set the trace info to “inter process communication”.
TraceData	=<MessageId><HookParameter0><HookParameter1><…>	Concatenate firstly the message ID (must be unique in ECU) and then all required parameter. The order of parameter has to be considered in the FIBEX file.
TraceDataLength	4-65535	At least the message ID has to be passed in the trace data.

Table 3-12 Recommended Parameter of Dlt_SendTraceMessage for VFB tracing

`TraceData` enables to send individual data. Normally it is used to send the parameter values of the hook function’s signature. The number of parameter can vary in number and type of each hook function. There is no specification which parameter comes first (except the message ID, which is always first). Thus the order of parameter can be freely chosen, but it has to be according the specified messages in the FIBEX file.

In Table 3-13 one VFB trace message is defined in the FIBEX file. Due to this data, the hook function has two parameters. The first is of type uint16 and the second is of type uint8. To interpret the receiving data correctly, the payload of VFB trace message must contain 4Byte message ID, 2Byte data of parameter one and 1Byte data of parameter two.

Section	Description
<!-- ECU --> <fx:ECUS> <fx:ECU ID="ECUI"> <ho:SHORT-NAME>ECU ID</ho:SHORT-NAME> <fx:MANUFACTURER-EXTENSION> <SW_VERSION>001.000.000</SW_VERSION> <APPLICATIONS> <APPLICATION> <APPLICATION_ID>VFBT</APPLICATION_ID> <APPLICATION_DESCRIPTION>VFB tracing</APPLICATION_DESCRIPTION> <CONTEXTS> <CONTEXT> <CONTEXT_ID>EXAM</CONTEXT_ID> <CONTEXT_DESCRIPTION>RteCallHook</CONTEXT_DESCRIPTION> </CONTEXT> </CONTEXTS> </APPLICATION> </APPLICATIONS>	In the example on the left, only application ID and the corresponding context ID of one VFB trace event is defined. The application ID (=VFBT) is fixed. The context ID depends on the context ID in the ECU.

Section	Description
</fx:MANUFACTURER-EXTENSION> </fx:ECU> </fx:ECUS>	
<!-- PDUS --> <fx:PDUS> <!-- Message 3 --> <!-- 1. Parameter --> <fx:PDU ID="PDU_3_0"> <ho:SHORT-NAME>PDU_3_0</ho:SHORT-NAME> <ho:DESC>Parameter_1_Description:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!-- 2. Parameter --> <fx:PDU ID="PDU_3_1"> <ho:SHORT-NAME>PDU_3_1</ho:SHORT-NAME> <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_3_0"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT16"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> <!-- 3. Parameter --> <fx:PDU ID="PDU_3_2"> <ho:SHORT-NAME>PDU_3_2</ho:SHORT-NAME> <ho:DESC>Parameter_2_Description:</ho:DESC> <fx:BYTE-LENGTH>0</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> </fx:PDU> <!-- 4. Parameter --> <fx:PDU ID="PDU_3_3"> <ho:SHORT-NAME>PDU_3_3</ho:SHORT-NAME> <fx:BYTE-LENGTH>1</fx:BYTE-LENGTH> <fx:PDU-TYPE>OTHER</fx:PDU-TYPE> <fx:SIGNAL-INSTANCES> <fx:SIGNAL-INSTANCE ID="S_3_1"> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> <fx:SIGNAL-REF ID-REF="S_UINT8"/> </fx:SIGNAL-INSTANCE> </fx:SIGNAL-INSTANCES> </fx:PDU> </fx:PDUS>	Define all PDUs (== Message ID) and their parameter. In the example on the left only DEM events are defined (message ID == 1). The DEM event sends 2 parameter values to DLT master, they are called “S_1_0” and “S_1_0”. To add more information in the message, both parameter values get another parameter with a describing string. Thus, “1. Parameter” provides the string “Event ID:”. “2. Parameter” provides the first received parameter with type uint16. “3. Parameter” provides the string “New Event Status:” and “4. Parameter” provides the corresponding parameter value from type uint8. More messages and/or parameter can be added if required.
<!-- FRAMES --> <fx:FRAMES> <!-- 1. Log and Trace Message --> <fx:FRAME ID="ID_3"> <ho:SHORT-NAME>ID_3</ho:SHORT-NAME> <fx:BYTE-LENGTH>3</fx:BYTE-LENGTH> <fx:FRAME-TYPE>OTHER</fx:FRAME-TYPE> <fx:PDU-INSTANCES> <fx:PDU-INSTANCE ID="P_3_0"> <fx:PDU-REF ID-REF="PDU_3_0"/> <fx:SEQUENCE-NUMBER>0</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_3_1"> <fx:PDU-REF ID-REF="PDU_3_1"/> <fx:SEQUENCE-NUMBER>1</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_3_2"> <fx:PDU-REF ID-REF="PDU_3_2"/> <fx:SEQUENCE-NUMBER>2</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> <fx:PDU-INSTANCE ID="P_3_3"> <fx:PDU-REF ID-REF="PDU_3_3"/> <fx:SEQUENCE-NUMBER>3</fx:SEQUENCE-NUMBER> </fx:PDU-INSTANCE> </fx:PDU-INSTANCES> <fx:MANUFACTURER-EXTENSION> <MESSAGE_TYPE>DLT_TYPE_NW_TRACE</MESSAGE_TYPE> <MESSAGE_INFO>DLT_NW_TRACE_IPC</MESSAGE_INFO>	Define frame with message ID == 3 (no other messages are used). The message type has to be DLT_TYPE_NW_TRACE and the message info has to be DLT_NW_TRACE_IPC.

Section	Description
<APPLICATION_ID>VFBT</APPLICATION_ID> <CONTEXT_ID>EXAM</CONTEXT_ID> <MESSAGE_SOURCE_FILE>Dlt.c</MESSAGE_SOURCE_FILE> <MESSAGE_LINE_NUMBER>1</MESSAGE_LINE_NUMBER> </fx:MANUFACTURER-EXTENSION> </fx:FRAME> </fx:FRAMES>	

Table 3-13 FIBEX file content for one VFB trace message

Caution
The parameter order in the payload of a trace/log messages have to be equal to the parameter order in the FIBEX file.

4 API Description

4.1 Type Definitions

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

Type Name	C-Type	Description	Value Range
Dlt_ConfigType	uint8	This type is currently not used by the DLT module. It exists to satisfy the needs of the Dlt_Init function.	n.a
Dlt_SessionIDType	uint32	This type describes the Session ID.	n.a.
Dlt_ApplicationIDType	uint8 [4]	This type describes the Application ID.	n.a.
Dlt_ContextIDType	uint8 [4]	This type describes the Context ID.	n.a.
Dlt_MessageIDType	uint32	This type describes the unique Message ID for a message.	n.a.
Dlt_MessageOptionsType	uint8	This type determines the message type (log or trace) and whether verbose mode is used or not.	n.a.
Dlt_MessageLogLevelType	Enum	This type describes the log level for each log message.	0..6
Dlt_MessageLogInfoType	Struct	This type describes the message log level, message options, context ID and application ID of a verbose or non-verbose message.	n.a.
Dlt_MessageTraceInfoType	Struct	This type describes the message trace status, message options, context ID and application ID of a verbose or non-verbose message	n.a
Dlt_MessageTraceType	Enum	This type describes the trace status for each trace message.	1..5
Dlt_ReturnType	Enum	This type describes the return values of the DLT services.	0..7

Table 4-1 Type definitions

4.2 Services provided by DLT

4.2.1 Dlt_InitMemory

Prototype	
`void Dlt_InitMemory (void)`	
Parameter	
`void`	-

Return code	
void	-
Functional Description	
The global data of DLT module is initialized by calling this function. This function must be called before Dlt_Init.	
Particularities and Limitations	
> This function is synchronous. > This function is non-reentrant.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-2 Dlt_InitMemory

4.2.2 Dlt_Init

Prototype	
void **Dlt_Init** (const Dlt_ConfigType* ConfigPtr)	
Parameter	
ConfigPtr	Pointer to a configuration structure for DLT module initialization. Currently this parameter is unused, and a NULL pointer can be passed.
Return code	
void	-
Functional Description	
The DLT module is initialized by calling this function. This function must be called before any other function of the DLT is called.	
Particularities and Limitations	
> This function is synchronous. > This function is non-reentrant. > This function shall be called after initialization of the communication drivers, interfaces and the XCP module.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-3 Dlt_Init

4.2.3 Dlt_MainFunction

Prototype	
void **Dlt_MainFunction** (void)	
Parameter	
void	-

Return code	
`void`	-
Functional Description	
The main function controls the state machine if PduR is used. Otherwise it is empty.	
Particularities and Limitations	
> The main function exists despite the requirement [Dlt468]. > This function is synchronous. > This function is non-reentrant. > This function shall be called cyclically by RTE.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-4 Dlt_MainFunction

4.2.4 Dlt_GetVersionInfo

Prototype	
`void` **`Dlt_GetVersionInfo`** `(Std_VersionInfoType* VersionInfo)`	
Parameter	
`VersionInfo`	Pointer to a RAM structure which is initialized with the version number of the DLT module.
Return code	
`void`	-
Functional Description	
This function writes the DLT module version into the structure referenced by the given pointer parameter.	
Particularities and Limitations	
> This function is only available if the preprocessor switch DLT_VERSION_INFO_API is set to STD_ON. > This function is non-reentrant. > This function is synchronous.	
Call context	
> No limitations	

Table 4-5 Dlt_GetVersionInfo

4.2.5 Dlt_DetForwardErrorTrace

Prototype	
`void Dlt_DetForwardErrorTrace ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId)`	
Parameter	
`ModuleId`	The module which called the function Det_ReportError
`InstanceId`	The instance of the calling module
`ApiId`	The identifier of the API where the DET error occurred
`ErrorId`	The number of the reported error
Return code	
`void`	-
Functional Description	
This function is called from within the context of the DET module function Det_ReportError.	
Particularities and Limitations	
> This function is non-reentrant.	
> This function is synchronous.	
Call context	
> The call context of the DET call. Every context in which BSW modules are called is possible.	

Table 4-6 Dlt_DetForwardErrorTrace

4.2.6 Dlt_DemTriggerOnEventStatus

Prototype	
`void Dlt_DemTriggerOnEventStatus ( Dem_EventIdType EventId, Dem_EventStatusExtendedType EventStatusOld, Dem_EventStatusExtendedType EventStatusNew)`	
Parameter	
`EventId`	Identification of an Event by assigned event number. The Event Number is configured in the Dem. > Min.: 1 (0: Indication of no Event or Failure) > Max.: Result of configuration of Event Numbers in Dem (Max is either 255 or 65535)

EventStatusOld	Extended event status before change
EventStatusNew	Detected / reported of event status
Return code	
void	-
Functional Description	
This service is provided by the Dem in order to call DLT upon status changes. It is possible to filter DemEvent status changes by setting DLT_DEM_EVENT_FILTERING_ENABLED to STD_ON. In this case the DLT module calls the Appl_DltDemEventFilterCbk() function. Hence the application has the chance to filter specific DEM Events IDs or DEM Event Status Bits. If the preprocessor switch DLT_DEM_EVENT_FILTERING_ENABLED is STD_OFF all changes of all DEM Events are reported by the DLT.	
Particularities and Limitations	
> The call to the function Dlt_DemTriggerOnEventStatus must be configured in the DEM module's configuration. > This function is non-reentrant. > This function is synchronous.	
Call context	
> The call context of the DEM Event status change.	

Table 4-7 Dlt_DemTriggerOnEventStatus

4.2.7 Dlt_SendLogMessage

Prototype

```
Dlt_ReturnType Dlt_SendLogMessage (
  Dlt_SessionIDType SessionId,
  const Dlt_MessageLogInfoType* LogInfo,
  const uint8* LogData,
  uint16 LogDataLength)
```

Parameter	
SessionId	For SW-C this is not visible (Port defined argument value), for BSW-modules it is the module number.
LogInfo	Structure containing information whether the message shall be transmitted in verbose or non-verbose mode and filtering information.
LogData	Buffer containing the parameters to be logged. The content of this pointer represents the payload of the send log message.
LogDataLength	Length of the data buffer LogData.

Return code	
Dlt_ReturnType	DLT_E_MSG_TOO_LARGE - The message is too large for sending over the network DLT_E_IF_NOT_AVAILABLE - The interface is not available. DLT_E_UNKNOWN_SESSION_ID - The provided session id is unknown. DLT_E_NOT_IN_VERBOSE_MODE - Unable to send the message in verbose mode. DLT_E_OK - The required operation succeeded.
Functional Description	
The service represents the interface to be used by basic software modules or by software component to send verbose and non-verbose log messages. For details how verbose or non-verbose DLT messages are transmitted refer to section 3.4.2.	
Particularities and Limitations	
> It is expected that the value LogInfo->logInfo is in valid range (DLT_LOG_FATAL – DLT_LOG_VERBOSE). There is no check, thus the value is passed to external tool. > The Message IDs used for DEM (0x00000001) and DET (0x00000002) are reserved and not usable for non-verbose log messages. > Logging with verbose messages is only possible if DLT_USE_VERBOSE_MODE is STD_ON. > Transmitting a mixture of static and non-static data is not supported for non-verbose messages. > This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-8 Dlt_SendLogMessage

4.2.8 Dlt_SendTraceMessage

Prototype

```
Dlt_ReturnType Dlt_SendTraceMessage (
  Dlt_SessionIDType SessionId,
  const Dlt_MessageTraceInfoType* TraceInfo,
  const uint8* TraceData,
  uint16 TraceDataLength)
```

Parameter	
SessionId	For SW-C this is not visible (Port defined argument value), for BSW-modules it is the module number. This parameter is specified by AUTOSAR but is currently not used by the implementation.
TraceInfo	Structure containing information whether the message shall be transmitted in verbose or non-verbose mode and filtering information.
TraceData	Buffer containing the parameters to be logged. The content of this pointer represents the payload of the send trace message.

TraceDataLength	Length of the data buffer TraceData.
Return code	
Dlt_ReturnType	DLT_E_MSG_TOO_LARGE - The message is too large for sending over the network DLT_E_IF_NOT_AVAILABLE - The interface is not available. DLT_E_UNKNOWN_SESSION_ID - The provided session id is unknown. DLT_E_NOT_IN_VERBOSE_MODE - Unable to send the message in verbose mode. DLT_E_OK - The required operation succeeded.
Functional Description	
The service represents the interface to be used by basic software modules or by software component to send verbose and non-verbose trace messages. For details how verbose or non-verbose DLT messages are transmitted refer to section 3.4.2.	
Particularities and Limitations	
> It is expected that the value TraceInfo->traceInfo is in valid range. The range depends on TraceInfo->options (DLT_TYPE_APP_TRACE || DLT_TYPE_NW_TRACE). There is no check, thus the value is passed to external tool. > The Message IDs used for DEM (0x00000001) and DET (0x00000002) are reserved and not usable for non-verbose log messages. > Tracing with verbose messages is only possible if DLT_USE_VERBOSE_MODE is STD_ON. > Transmitting a mixture of static and non-static data is not supported for non-verbose messages. > This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-9 Dlt_SendTraceMessage

4.2.9 Dlt_RegisterContext

Prototype

```
Dlt_ReturnType Dlt_RegisterContext(
    Dlt_SessionIDType SessionId,
    const uint8* AppId,
    const uint8* ContextId,
    const uint8* AppDescription,
    uint8 LenAppDescription,
    const uint8* ContextDescription,
    uint8 LenContextDescription
)
```

Parameter	
SessionId	Number of the module (Module ID within BSW, Port defined argument value within SW-C)
AppId	The Application ID.
ContextId	The Context ID
AppDescription	Points to description string for the provided application id. At maximum 255 characters are interpreted.
LenAppDescription	The length of the description for the application id string (number of characters of description string).
ContextDescription	Points to description string for the provided context. At maximum 255 characters are interpreted.
LenContextDescription	The length of the description string (number of characters of description string).
Return code	
Dlt_ReturnType	DLT_E_CONTEXT_ALREADY_REG - The software module context has already been registered. DLT_E_ERROR_TO_MANY_CONTEXT – There were too many registrations of contexts for one application ID, too many registrations of application IDs or too many registrations of overall context IDs. DLT_E_UNKNOWN_SESSION_ID - The session ID is not valid. DLT_E_OK - The required operation succeeded.
Functional Description	
Registers the DLT users' context. The service must be called when a software module wants to use services offered by DLT software component for a specific context. If a context id is being registered for an already registered application id then AppDescription can be NULL and LenAppDescription zero.	
Particularities and Limitations	
> This function is not reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-10 Dlt_RegisterContext

4.2.10 Dlt_SetState

Prototype	
`Std_ReturnType **Dlt_SetState** (Dlt_GlobalStateType NewState)`	
Parameter	
NewState	The new global DLT state.

Return code	
Std_ReturnType	E_OK: The global DLT state change was successfully requested. E_NOT_OK: The change request was rejected.
Functional Description	
The service dis-/enables the DLT. In global DLT state DLT_GLOBAL_STATE_OFFLINE, most DLT services are not available. Whereas in global DLT state DLT_GLOBAL_STATE_ONLINE all DLT services are available.	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This service should be called in context of a diagnostic session.	

Table 4-11 Dlt_SetState

4.2.11 Dlt_GetState

Prototype	
`Std_ReturnType Dlt_GetState (Dlt_GlobalStateType* CurrentStatePtr)`	
Parameter	
CurrentStatePtr	The requested global DLT state.
Return code	
Std_ReturnType	E_OK: The return of current global DLT state was successfully. E_NOT_OK: The current global DLT state could not be returned.
Functional Description	
The service returns the current global DLT state.	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This service should be called in context of a diagnostic session.	

Table 4-12 Dlt_GetState

4.2.12 Dlt_SetLogLevel

Prototype

`Std_Returntype Dlt_SetLogLevel (`
`  const uint8* AppId,`
`  const uint8* ContextId,`
`  Dlt_MessageLogLevelType NewLogLevel)`

Parameter	
AppId	The DLT user (SWC) for which the log level changes.
ContextId	The context of the DLT user for which the log level changes.
NewLogLevel	The new log level for the DLT user(s).
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Notifies the DLT to change the log level of the specified DLT user(s).	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-13 Dlt_SetLogLevel

4.2.13 Dlt_SetTraceStatus

Prototype	
Std_Returntype **Dlt_SetTraceStatus** (const uint8* AppId, const uint8* ContextId, boolean NewTraceStatus)	
Parameter	
AppId	The DLT user (SWC) for which the trace status changes.
ContextId	The context of the DLT user for which the trace status changes.
NewTraceStatus	The new trace status for the DLT user(s).
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Notifies the DLT to change the trace status of the specified DLT user.	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-14 Dlt_SetTraceStatus

4.2.14 Dlt_SetDefaultLogLevel

Prototype	
`Std_ReturnType` **`Dlt_SetDefaultLogLevel`** `(Dlt_MessageLogLevelType NewLogLevel)`	
Parameter	
`NewLogLevel`	The new default log level.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Notifies the DLT to change the default log level.	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-15 Dlt_SetDefaultLogLevel

4.2.15 Dlt_GetDefaultLogLevel

Prototype	
`Std_ReturnType` **`Dlt_GetDefaultLogLevel`** `(Dlt_MessageLogLevelType *DefaultLogLevel)`	
Parameter	
`DefaultLogLevel`	The current default log level.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the current default log level.	
Particularities and Limitations	
> This function is non-reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-16 Dlt_GetDefaultLogLevel

4.2.16 Dlt_SetDefaultTraceStatus

Prototype	
`Std_ReturnType Dlt_SetDefaultTraceStatus ( boolean NewTraceStatus, const uint8* LogChannelName)`	
Parameter	
`NewTraceStatus`	The new trace status for the DLT user(s).
`LogChannelName`	**Notes:** 1. The parameter is optional (available if "`/MICROSAR/Dlt/DltGeneral/DltSupportDefaultAPIs`" is enabled) 2. Parameter not supported!
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Notifies the DLT to change the default trace status.	
Particularities and Limitations	
> This function is non-reentrant.	
> This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-17 Dlt_SetDefaultTraceStatus

4.2.17 Dlt_GetDefaultTraceStatus

Prototype	
`Std_ReturnType Dlt_GetDefaultTraceStatus ( const uint8* LogChannelName, boolean* TraceStatus)`	
Parameter	
`LogChannelName`	**Notes:** 1. The parameter is optional (available if "`/MICROSAR/Dlt/DltGeneral/DltSupportDefaultAPIs`" is enabled) 2. Parameter not supported!
`TraceStatus`	The new trace status for the DLT user(s).
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the current default trace status.	

Particularities and Limitations
> This function is non-reentrant.
> This function is synchronous.
Call context
> This function can be called in any context.

Table 4-18 Dlt_GetDefaultTraceStatus

4.2.18 Dlt_SetMessageFiltering

Prototype	
`Std_ReturnType` **`Dlt_SetMessageFiltering`** `(boolean Status)`	
Parameter	
`Status`	The new status for message filtering. FALSE: Messages are not filtered. TRUE: Messages are filtered by their log level or trace status.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Notifies the DLT to change the message filter behavior.	
Particularities and Limitations	
> This function is only available if the preprocessor switch DLT_IMPLEMENT_FILTER_MESSAGES is available.	
> This function is non-reentrant.	
> This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-19 Dlt_SetMessageFiltering

4.2.19 Dlt_ResetToFactoryDefault

Prototype	
`Std_ReturnType` **`Dlt_ResetToFactoryDefault`** `(void)`	
Parameter	
`void`	N/A
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.

Functional Description
Notifies the DLT to reset all changed configuration to the default values.
Particularities and Limitations
> This function is reentrant. > This function is synchronous.
Call context
> This function can be called in any context.

Table 4-20 Dlt_ResetToFactoryDefault

4.2.20 Dlt_StoreConfiguration

Prototype	
`Std_ReturnType` **`Dlt_StoreConfiguration`** `(void)`	
Parameter	
`void`	N/A
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed. DLT_E_NOT_SUPPORTED: The service is not supported.
Functional Description	
Notifies the DLT to store the current DLT configuration to non-volatile memory.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-21 Dlt_StoreConfiguration

4.2.21 Dlt_GetLogInfo

Prototype
`Std_ReturnType` **`Dlt_GetLogInfo`** `(uint8 Options, uint8* AppId, uint8* ContextId, uint8 Status, Dlt_LogInfoType* LogInfo)`

Parameter	
Options	Defines which data is requested: - DLT_LOG_OPTIONS_ALL_INFO_WITHOUT_DESCRIPTIONS (= 6): - DLT_LOG_OPTIONS_ALL_INFO_WITH_DESCRIPTIONS (= 7) Note: Because descriptions are not supported, request is in both cases handled equally. Therefore, in both cases the application id, context id, log level, and the trace status of requested Dlt user is returned. All other values result in return value E_NOT_OK.
AppId	The application id to identify the Dlt user. If set to 0 (= wildcard), information of all registered Dlt users is requested.
ContextId	The context id to identify the Dlt user. If set to 0 (= wildcard), information of all contexts registered under given application id is requested.
Status	Not used parameter.
LogInfo	Contains the requested information.
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the requested Dlt user information.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-22 Dlt_GetLogInfo

4.2.22 Dlt_GetTraceStatus

Prototype	
`Std_ReturnType` **`Dlt_GetTraceStatus`** `(uint8* AppId, uint8* ContextId, boolean* TraceStatus)`	
Parameter	
AppId	The application id to identify the Dlt user.
ContextId	The context id to identify the Dlt user.
TraceStatus	Pointer to the trace status.
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.

Functional Description
Returns the trace status of the requested Dlt user. Note: Wildcards are not supported.
Particularities and Limitations
> This function is reentrant. > This function is synchronous.
Call context
> This function can be called in any context.

Table 4-23 Dlt_GetTraceStatus

4.2.23 Dlt_GetLogChannelNames

Prototype	
`Std_ReturnType` **`Dlt_GetLogChannelNames`** `(uint8* NumberOfLogChannels, uint8* LogChannelNames)`	
Parameter	
`NumberOfLogChannels`	Input and output parameter. As input parameter: defines how many log channel names are requested. As output parameter: defines how many log channels are configured/available. Note: can be used with 0, then no log channel name is reported, but the caller is informed how many log channels are available.
`LogChannelNames`	Dlt copies a string of all concatenated log channel names. Each log channel name has exactly a 4 bytes length. If a log channel name is shorter than 4 bytes (e.g. “LC1”), the missing byte(s) are filled with the value 0x00.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the log channel names of requested log channels. Because it is only possible to request a number of log channel names and not the name of an explicit log channel, the order of returned log channel names is always according to the index of the log channels. Example: There are three log channels. First log channel is called “LCh0”, the second is called “LCh1” and the third is called “LCh2”. If one log channel name is requested, ‘`LogChannelNames`’ will be set to “LCh0”. If all log channel names are requested, ‘`LogChannelNames`’ will be set to “LCh0LCh1LCh2”.	

Particularities and Limitations
> This function is reentrant.
> This function is synchronous.
Call context
> This function can be called in any context.

Table 4-24 Dlt_GetLogChannelNames

4.2.24 Dlt_GetLogChannelThreshold

Prototype	
`Std_ReturnType` **`Dlt_GetLogChannelThreshold`** `(uint8* LogChannelName, Dlt_MessageLogLevelType* Threshold, boolean* TraceStatus)`	
Parameter	
`LogChannelName`	The log channel name used as identifier.
`Threshold`	Pointer which is set to the current log channel threshold.
`TraceStatus`	Pointer which is set to the current log channel trace status.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the threshold and the trace status of the requested log channel. Note: The log channel specific threshold and trace status are used as additional filter values for log and trace messages. A log or trace message may be assigned to multiple log channels, but it is possible that it is only transmitted via some of the assigned log channels, because of the log channel threshold and trace status.	
Particularities and Limitations	
> This function is reentrant.	
> This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-25 Dlt_GetLogChannelThreshold

4.2.25 Dlt_SetLogChannelThreshold

Prototype	
`Std_ReturnType` **`Dlt_SetLogChannelThreshold`** `(uint8* LogChannelName, Dlt_MessageLogLevelType NewThreshold, boolean NewTraceStatus)`	
Parameter	
`LogChannelName`	The log channel name used as identifier.

NewThreshold	The new value for the log channel threshold.
NewTraceStatus	The new value for the log channel trace status.
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Sets the threshold and the trace status of the requested log channel to new values.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-26 Dlt_SetLogChannelThreshold

4.2.26 Dlt_SetLogChannelAssignment

Prototype	
`Std_ReturnType Dlt_SetLogChannelAssignment (uint8* AppId, uint8* ContextId, uint8* LogChannelName, Dlt_AsignmentOperation AddRemoveOp)`	
Parameter	
AppId	The application id to identify the Dlt user.
ContextId	The context id to identify the Dlt user.
LogChannelName	The log channel name used as identifier.
AddRemoveOp	Valid values: - DLT_ASSIGN_ADD: Add the log channel to given Dlt user. - DLT_ASSIGN_REMOVE: Remove the log channel from given Dlt user.
Return code	
Std_ReturnType	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Adds or removes a log channel assignment of a Dlt user.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-27 Dlt_SetLogChannelAssignment

4.2.27 Dlt_InjectCall_<SessionId>

Prototype	
`void Dlt_InjectCall_<SessionId> (uint8* AppId, uint8* ContextId, uint32 ServiceId, uint32 DataLength, uint8* Data)`	
Parameter	
`AppId`	The application id to identify the target SWC (where the service shall be injected).
`ContextId`	The context id to identify the target SWC (where the service shall be injected).
`ServiceId`	The service to be executed by the SWC.
`DataLength`	The length of data required to execute the injected service.
`Data`	The data required to execute the injected service.
Return code	
`void`	N/A
Functional Description	
Forwards the injection request to the generic API Dlt_InjectCall.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-28 Dlt_InjectCall_<SessionId>

4.2.28 Dlt_InjectCall

Prototype	
`void Dlt_InjectCall (uint32 SessionId, uint8* AppId, uint8* ContextId, uint32 ServiceId, uint32 DataLength, uint8* Data)`	
Parameter	
`SessionId`	The session id to identify the target SWC (where the service shall be injected).
`AppId`	The application id to identify the target SWC (where the service shall be injected).
`ContextId`	The context id to identify the target SWC (where the service shall be injected).
`ServiceId`	The service to be executed by the SWC.
`DataLength`	The length of data required to execute the injected service.
`Data`	The data required to execute the injected service.
Return code	
`void`	N/A
Functional Description	
Forwards the injection request to corresponding SWC.	

Particularities and Limitations
> This function is reentrant. > This function is synchronous.
Call context
> This function can be called in any context.

Table 4-29 Dlt_InjectCall

4.2.29 Dlt_GetLogChannelDebugMode

Prototype	
`Std_ReturnType` **`Dlt_GetLogChannelDebugMode`** `(uint8* LogChannelName, Dlt_DebugModeType *DebugModePtr)`	
Parameter	
`LogChannelName`	The log channel name used as identifier.
`DebugModePtr`	Pointer which is set to the current debug mode of given log channel.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed.
Functional Description	
Returns the debug mode of the requested log channel.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-30 Dlt_GetLogChannelDebugMode

4.2.30 Dlt_SetLogChannelDebugMode

Prototype	
`Std_ReturnType` **`Dlt_SetLogChannelDebugMode`** `(uint8* LogChannelName, Dlt_DebugModeType NewDebugMode)`	
Parameter	
`LogChannelName`	The log channel name used as identifier.
`NewDebugMode`	The new debug mode for the log channel threshold: - DLT_DEBUGMODE_CONTINUOUS_FIRSTISBEST - DLT_DEBUGMODE_ONEVENT_FIRSTISBEST

Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed. DLT_E_DEBUG_MODE_ALREADY_ACTIVE: The requested debug mode is already active. DLT_E_DEBUG_MODE_ALREADY_REQUESTED: The requested debug mode is already requested, but not activated yet.
Functional Description	
Requests new debug mode of the requested log channel. The actual debug mode change is done in next call of Dlt_MainFunction.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-31 Dlt_SetLogChannelDebugMode

4.2.31 Dlt_TriggerLogChannelDebugEvent

Prototype	
`Std_ReturnType` **`Dlt_TriggerLogChannelDebugEvent`** `(uint8* LogChannelName)`	
Parameter	
`LogChannelName`	The log channel name used as identifier.
Return code	
`Std_ReturnType`	E_OK: The request succeeded. E_NOT_OK: The request failed. DLT_E_DEBUG_MODE_WRONG: The currently active debug mode does not support debug events. DLT_E_PENDING: A debug mode change or a previous debug event is currently active, therefore try again later.
Functional Description	
Triggers a debug event on the requested log channel. In debug mode ‘DLT_DEBUGMODE_ONEVENT_FIRSTISBEST’, this event starts the buffering.	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in any context.	

Table 4-32 Dlt_TriggerLogChannelDebugEvent

4.2.32 Dlt_OsVthSchedule

Prototype	
`void Dlt_OsVthSchedule (uint32 FromThreadId, uint32 FromThreadReason, uint32 ToThreadId, uint32 ToThreadReason, uint16 CallerCoreId)`	
Parameter	
`FromThreadId`	The thread id just terminated, suspended, or preempted. The thread id consists of all task ids (0 – n-1, where n is the number of tasks) and all category 2 ISRs (n – m-1, where m-n is the number of category 2 ISRs).
`FromThreadReason`	The reason why the context of `FromThreadId` was left.
`ToThreadId`	The thread id just activated or resumed.
`ToThreadReason`	The reason why the context of `ToThreadId` was entered.
`CallerCoreId`	The core id on which the context switch is performed.
Return code	
`void`	N/A
Functional Description	
This API shall only be called by MSR OS timing hook OS_VTH_SCHEDULE. This API uses the received parameter to create a Dlt trace message in non-verbose mode. This message is buffered in all assigned log channels. Per default, the message is only transmitted via default log channel (/MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltDefaultLogChannelRef).	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in OS context.	

Table 4-33 Dlt_OsVthSchedule

4.2.33 Dlt_OsVthActivation

Prototype	
`void Dlt_OsVthActivation (uint32 TaskId, uint16 DestCoreId, uint16 CallerCoreId)`	
Parameter	
`TaskId`	The task id just activated.
`DestCoreId`	Not used.
`CallerCoreId`	The core id on which the task is activated.
Return code	
`void`	N/A

Functional Description
This API shall only be called by MSR OS timing hook OS_VTH_ACTIVATION. This API uses the received parameter to create a Dlt trace message in non-verbose mode. This message is buffered in all assigned log channels. Per default, the message is only transmitted via default log channel (/MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltDefaultLogChannelRef).
Particularities and Limitations
> This function is reentrant. > This function is synchronous.
Call context
> This function can be called in OS context.

Table 4-34 Dlt_OsVthActivation

4.2.34 Dlt_OsVthSetEvent

Prototype	
`void Dlt_OsVthSetEvent (uint32 TaskId, Boolean StateChanged, uint16 DestCoreId, uint16 CallerCoreId)`	
Parameter	
`TaskId`	The task id just activated.
`StateChanged`	
`DestCoreId`	Not used.
`CallerCoreId`	The core id on which the task is activated.
Return code	
`void`	N/A
Functional Description	
This API shall only be called by MSR OS timing hook OS_VTH_SETEVENT. This API uses the received parameter to create a Dlt trace message in non-verbose mode. This message is buffered in all assigned log channels. Per default, the message is only transmitted via default log channel (/MICROSAR/Dlt/DltConfigSet/DltLogOutput/DltDefaultLogChannelRef).	
Particularities and Limitations	
> This function is reentrant. > This function is synchronous.	
Call context	
> This function can be called in OS context.	

Table 4-35 Dlt_OsVthSetEvent

4.3 Services used by DLT

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

Component	API
DET	Det_ReportError
XCP	Xcp_Event
SCHM	SchM_Enter_Dlt SchM_Exit_Dlt
APPL	Appl_DltDemEventFilterCbk
GPT	Gpt_GetTimeElapsed
NvM	NvM_InvalidateNvBlock NvM_WriteBlock NvM_GetErrorStatus
StbM	StbM_GetCurrentTime

Table 4-36 Services used by the DLT

5 Glossary and Abbreviations

5.1 Glossary

Term	Description
Cfg5	Generation tool for MICROSAR components
DltViewer	DLT master tool of GENIVI.

Table 5-1 Glossary

5.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CD	Complex Driver
DEM	Diagnostic Event Manager
DET	Development Error Tracer
DLT	Diagnostic Log and Trace
ECU	Electronic Control Unit
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
RTE	Runtime Environment
SWC	Software Component
SWS	Software Specification
VFB	Virtual Function Bus

Table 5-2 Abbreviations

6 Contact

Visit our website for more information on

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

www.vector.com