Document Title	Specification of Log and Trace
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	853

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R23-11

Document Change History

Date	Release	Changed by	Description
2023-11-23	R23-11	AUTOSAR Release Management	• AP Tracing with modeled Messages incorporated • Dependencies to other Functional Clusters • Interface for sending log messages added
2022-11-24	R22-11	AUTOSAR Release Management	• Specification of modeled messages finished • InstanceSpecifier introduced • Introduced optional fragmentation (segmentation) header • Privacy flags and message tags added
2021-11-25	R21-11	AUTOSAR Release Management	• Removed useless exceptionalsafety elements • Provided a logmode to logchannel mapping configuration possibility • Header files, c++ syntax errors clean-up
2020-11-30	R20-11	AUTOSAR Release Management	• Introduced `Non-modeled messages` and `Modeled messages` to Chapter 7.3 Log Messages • Introduced `Logger::WithLevel()` API, to log messages and pass the LogLevel as an API parameter • Refactoring and editorial changes

△

2019-11-28	R19-11	AUTOSAR Release Management	• Removed Class LogManager. Moved `remoteClientState()` to Chapter 8.2 Function definitions (logging.h) • Added Functional Cluster shutdown behavior. Added Funtional Cluster initialization via `ara::core::Initialize()` • Removed TSYNC related spec items from Chapter 7.4 • Refactoring and editorial changes • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Changed APIs (Logstream, Logmanager, Logging) • Refactoring and editorial changes
2018-10-31	18-10	AUTOSAR Release Management	• Changed initialization APIs • Improved references • Log file definition
2018-03-29	18-03	AUTOSAR Release Management	• Refactoring and editorial changes • Log and Trace extensions added
2017-10-27	17-10	AUTOSAR Release Management	• No content changes
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Contents

References

[1] Log and Trace Protocol Specification
AUTOSAR_FO_PRS_LogAndTraceProtocol

[2] Glossary
AUTOSAR_FO_TR_Glossary

[3] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[4] General Requirements specific to Adaptive Platform
AUTOSAR_AP_RS_General

[5] Specification of Adaptive Platform Core
AUTOSAR_AP_SWS_Core

[6] Explanation of Adaptive Platform Software Architecture
AUTOSAR_AP_EXP_SWArchitecture

[7] Requirements on Log and Trace
AUTOSAR_FO_RS_LogAndTrace

[8] Specification of Time Synchronization
AUTOSAR_AP_SWS_TimeSynchronization

[9] Specification of Operating System Interface
AUTOSAR_AP_SWS_OperatingSystemInterface

1 Introduction and functional overview

This specification specifies the functionality of the AUTOSAR Adaptive Platform Log and Trace.

The Log and Trace provides interfaces for Adaptive Applications to forward logging information onto the communication bus, the console, to the file system or to ARTI-trace. Each of the provided logging information has its own severity level. For each severity level, a separate method is provided to be used by applications or Adaptive Platform Services, e.g. ara::com. In addition, utility methods are provided to convert decimal values into the hexadecimal numeral system, or into the binary numeral system.

To pack the provided logging information into a standardized delivery and presentation format, a protocol is needed. For this purpose, the LT protocol can be used, which is standardized within the AUTOSAR consortium.

The LT protocol can add additional information to the provided logging information. This information can be used by a Logging client to relate, sort or filter the received logging frames.

Detailed information regarding the use cases and the LT protocol itself are provided by the PRS Log and Trace protocol specification. For more information regarding the LT protocol refer to [1].

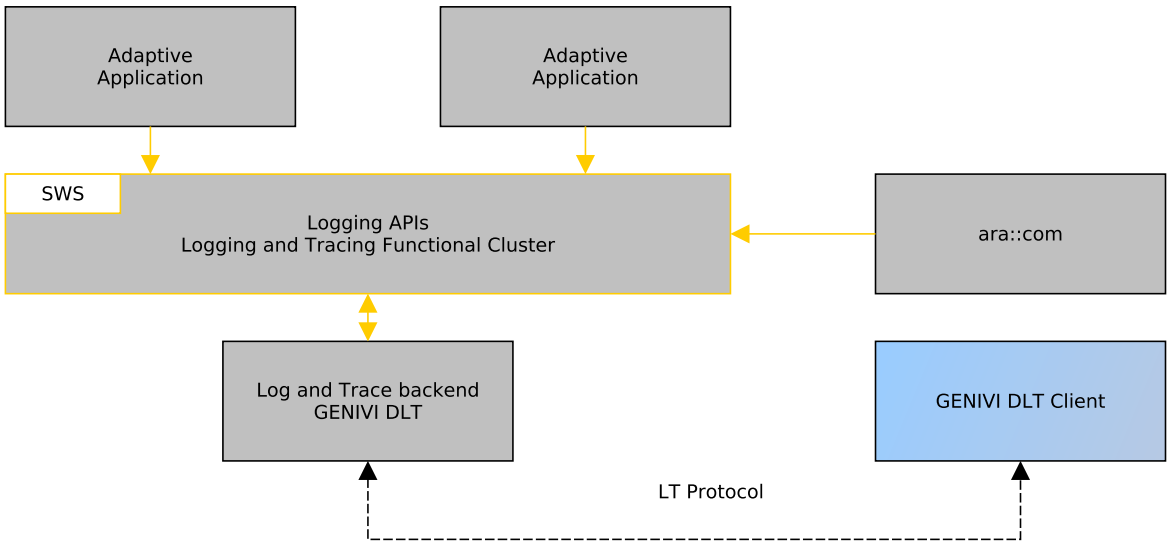

Figure 1.1: Architecture overview

The ARTI-trace methods are integrated in the ara::log Functional Cluster. The support of ARTI-trace functionality of non AUTOSAR applications or services is handled by adapters (i.e. “OS/ara::log Adapter”).

Furthermore, this document introduces additional specification extensions for the AUTOSAR Adaptive Platform Log and Trace.

2 Acronyms and Abbreviations

The following table contains the list of terms and abbreviations used in the scope of this document which are not already defined in [2, AUTOSAR glossary]. along with the spelled-out meaning of each of the abbreviations.

Abbreviation / Acronym:	Description:
Log and Trace	The official `Functional Cluster` name that manages the logging
L&T	Acronym for `Log and Trace`
LT protocol	Original name of the protocol itself (Log and Trace), specified in the PRS document [1]
Logging API	The main logging interface towards user applications as a library
Logging back-end	Implementation of the `LT protocol`, e.g. `DLT`
Logging Client	An external tool which can remotely interact with the `Logging framework`
Logging framework	Implementation of the software solution used for logging purposes
Logging instance	The class that enables the logging functionality and handles a single logging context
Log message	Log message, including message header(s)
Log severity level	Meta information about the severity of a passed logging information
DLT	Diagnostics Log and Trace - a GENIVI Log and Trace daemon implementation of the `LT protocol`
Application process	An executable instance (process) that is running on a `Machine`
ara::log Modeled Message Generator	A workflow tool (e.g. a script) with the purpose to read-parse an ARXML model of data types in an Adaptive Platform Interface and generate a corresponding modeled message specific representation thereof.
ARTI-trace	Trace capabilies that are provided by the system, i.e. LTTNG (Linux Trace Toolkit Next Generation) or hardware tracer, see also `ARTI`

The following technical terms used throughout this document are defined in the official [2] AUTOSAR Glossary or [3] TPS Manifest Specification – they are repeated here for tracing purposes.

Term	Description
Adaptive Application	see [2] AUTOSAR Glossary
Application	see [2] AUTOSAR Glossary
AUTOSAR Adaptive Platform	see [2] AUTOSAR Glossary
Adaptive Platform Foundation	see [2] AUTOSAR Glossary
Manifest	see [2] AUTOSAR Glossary
Executable	see [2] AUTOSAR Glossary
Functional Cluster	see [2] AUTOSAR Glossary
Adaptive Platform Service	see [2] AUTOSAR Glossary
Machine	see [2] AUTOSAR Glossary
Service	see [2] AUTOSAR Glossary
Service Interface	see [2] AUTOSAR Glossary
Service Discovery	see [2] AUTOSAR Glossary
ARTI	see [2] AUTOSAR Glossary

Table 2.1: Glossary-defined Technical Terms

3 Input documents & related standards and norms

3.1 Input documents

NOTE: [4, RS-RSGeneral] is listed here as an input document because it applies to SWS LogAndTrace as well as to all SWS documents of the Adaptive Platform. Since it includes only non-functional requirements the tracing is not necessary.

3.2 Further applicable specification

AUTOSAR provides a core specification [5] which is also applicable for Log and Trace. The chapter “General requirements for all Functional Clusters” of this specification shall be considered as an additional and required specification for implementation of Log and Trace.

4 Constraints and assumptions

4.1 Known limitations

The provided Logging framework API is designed to be independent from the underlying Logging back-end implementation and as such doesn't impose limitations.

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other Functional Clusters

This chapter provides an overview of the dependencies to other Functional Clusters in the AUTOSAR Adaptive Platform. Section 5.1 “Provided Interfaces” lists the interfaces provided by `Log and Trace` to other Functional Clusters. Section 5.2 “Required Interfaces” lists the interfaces required by `Log and Trace`.

A detailed technical architecture documentation of the AUTOSAR Adaptive Platform is provided in [6].

5.1 Provided Interfaces

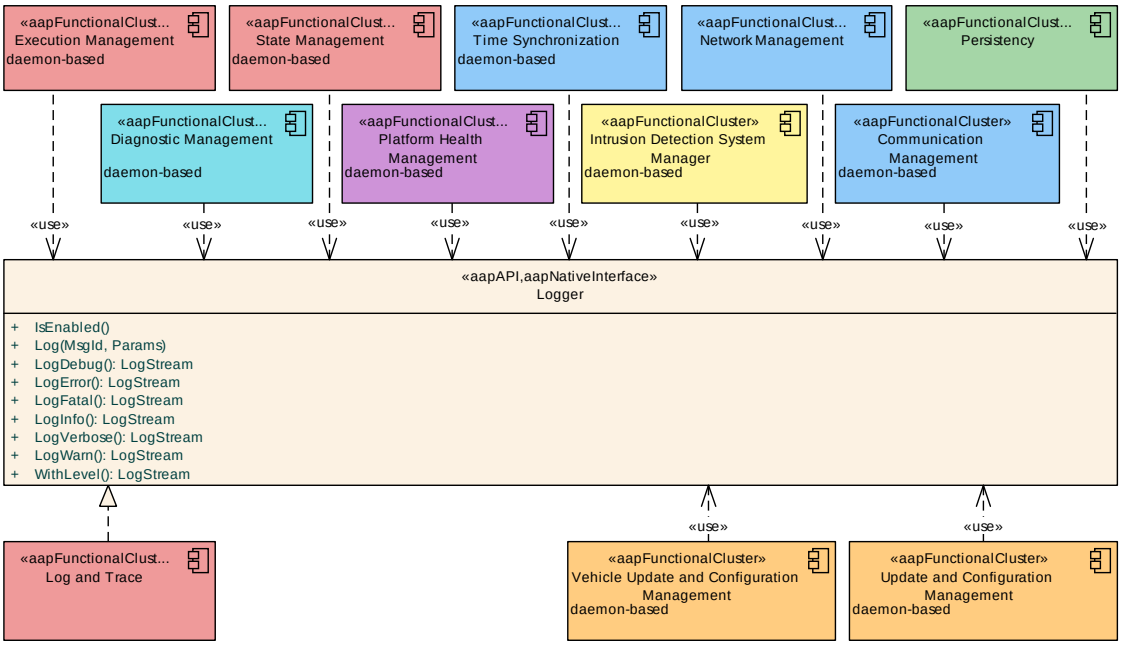

Figure 5.1: Interfaces provided by Log and Trace to other Functional Clusters

Figure 5.1 shows the interfaces provided by `Log and Trace` to other Functional Clusters within the AUTOSAR Adaptive Platform. Table 5.1 provides a complete list of interfaces provided to other Functional Clusters within the AUTOSAR Adaptive Platform.

Interface	*Functional Cluster*	*Purpose*
`Logger`	Communication Management	`Communication Management` shall use this interface to log standardized messages.
	Diagnostic Management	`Diagnostic Management` shall use this interface to log standardized messages.
	Execution Management	`Execution Management` shall use this interface to log standardized messages.
	Intrusion Detection System Manager	`Adaptive Intrusion Detection System Manager` shall use this interface to log standardized messages.

△

Interface	*Functional Cluster*	*Purpose*
	Network Management	`Network Management` shall use this interface to log standardized messages.
	Persistency	`Persistency` shall use this interface to log standardized messages.
	Platform Health Management	`Platform Health Management` shall use this interface to log standardized messages.
	Raw Data Stream	`Raw Data Stream` uses this interface to log standardized messages.
	State Management	`State Management` shall use this interface to log standardized messages.
	Time Synchronization	`Time Synchronization` shall use this interface to log standardized messages.
	Update and Configuration Management	`Update and Configuration Management` shall use this interface to log standardized messages.
	Vehicle Update and Configuration Management	`Vehicle Update and Configuration Management` shall use this interface to log standardized messages.

Table 5.1: Interfaces provided to other Functional Clusters

5.2 Required Interfaces

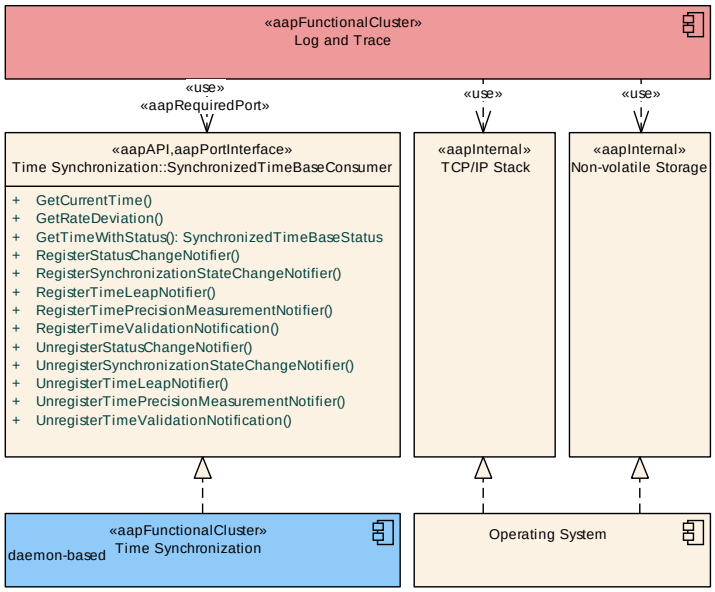

Figure 5.2: Interfaces required by Log and Trace from other Functional Clusters

Figure 5.2 shows the interfaces required by `Log and Trace` from other Functional Clusters within the AUTOSAR Adaptive Platform. Table 5.2 provides a complete list of required interfaces from other Functional Clusters within the AUTOSAR Adaptive Platform.

Functional Cluster	*Interface*	*Purpose*
Time Synchronization	SynchronizedTimeBaseConsumer	`Log and Trace` shall use this interface to determine the timestamps that are associated with log messages.

Table 5.2: Interfaces required from other Functional Clusters

6 Requirements Tracing

The following table references the requirements specified in RS Log And Trace [7] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_LT_00002]		[SWS_LOG_00227] [SWS_LOG_00228] [SWS_LOG_00229] [SWS_LOG_00230] [SWS_LOG_00231] [SWS_LOG_00232] [SWS_LOG_00233] [SWS_LOG_00234] [SWS_LOG_00235] [SWS_LOG_00236] [SWS_LOG_00237] [SWS_LOG_00238]
[RS_LT_00003]		[SWS_LOG_00001] [SWS_LOG_00002] [SWS_LOG_00004] [SWS_LOG_00005] [SWS_LOG_00006] [SWS_LOG_00007] [SWS_LOG_00008] [SWS_LOG_00009] [SWS_LOG_00010] [SWS_LOG_00011] [SWS_LOG_00012] [SWS_LOG_00013] [SWS_LOG_00018] [SWS_LOG_00021] [SWS_LOG_00039] [SWS_LOG_00040] [SWS_LOG_00041] [SWS_LOG_00042] [SWS_LOG_00043] [SWS_LOG_00044] [SWS_LOG_00045] [SWS_LOG_00046] [SWS_LOG_00047] [SWS_LOG_00048] [SWS_LOG_00049] [SWS_LOG_00050] [SWS_LOG_00051] [SWS_LOG_00062] [SWS_LOG_00063] [SWS_LOG_00064] [SWS_LOG_00065] [SWS_LOG_00066] [SWS_LOG_00067] [SWS_LOG_00068] [SWS_LOG_00069] [SWS_LOG_00070] [SWS_LOG_00082] [SWS_LOG_00083] [SWS_LOG_00091] [SWS_LOG_00095] [SWS_LOG_00098] [SWS_LOG_00123] [SWS_LOG_00124] [SWS_LOG_00128] [SWS_LOG_00129] [SWS_LOG_00130] [SWS_LOG_00131] [SWS_LOG_00132] [SWS_LOG_00133] [SWS_LOG_00172] [SWS_LOG_00173] [SWS_LOG_00201] [SWS_LOG_00203] [SWS_LOG_00204] [SWS_LOG_00205] [SWS_LOG_00240] [SWS_LOG_00241] [SWS_LOG_00242] [SWS_LOG_00244] [SWS_LOG_00245] [SWS_LOG_00246] [SWS_LOG_00247] [SWS_LOG_00248] [SWS_LOG_00249] [SWS_LOG_00250] [SWS_LOG_00251] [SWS_LOG_00253] [SWS_LOG_00254] [SWS_LOG_00256] [SWS_LOG_00257] [SWS_LOG_00258] [SWS_LOG_00261] [SWS_LOG_00263] [SWS_LOG_20000]
[RS_LT_00017]		[SWS_LOG_00083]
[RS_LT_00025]		[SWS_LOG_00236]
[RS_LT_00030]		[SWS_LOG_00095]
[RS_LT_00045]		[SWS_LOG_00007]

△

Requirement	Description	Satisfied by
[RS_LT_00046]		[SWS_LOG_00206] [SWS_LOG_00207] [SWS_LOG_00208] [SWS_LOG_00209] [SWS_LOG_00210] [SWS_LOG_00211] [SWS_LOG_00212] [SWS_LOG_00213] [SWS_LOG_00214] [SWS_LOG_00215] [SWS_LOG_00216] [SWS_LOG_00217] [SWS_LOG_00218] [SWS_LOG_00219] [SWS_LOG_00220] [SWS_LOG_00221] [SWS_LOG_00222] [SWS_LOG_00223] [SWS_LOG_00224] [SWS_LOG_00225] [SWS_LOG_00226]
[RS_LT_00047]		[SWS_LOG_00004]
[RS_LT_00048]		[SWS_LOG_00004]
[RS_LT_00049]		[SWS_LOG_00008] [SWS_LOG_00009] [SWS_LOG_00010] [SWS_LOG_00011] [SWS_LOG_00012] [SWS_LOG_00013] [SWS_LOG_00125] [SWS_LOG_00126] [SWS_LOG_00127] [SWS_LOG_00130] [SWS_LOG_00240] [SWS_LOG_00241] [SWS_LOG_00242] [SWS_LOG_00244] [SWS_LOG_00245] [SWS_LOG_00246] [SWS_LOG_00247] [SWS_LOG_00248] [SWS_LOG_00249] [SWS_LOG_00250] [SWS_LOG_00251] [SWS_LOG_00255] [SWS_LOG_00257]
[RS_LT_00050]		[SWS_LOG_00005] [SWS_LOG_00006] [SWS_LOG_00253] [SWS_LOG_00254]
[RS_LT_00052]		[SWS_LOG_00001]
[RS_LT_00056]		[SWS_LOG_00229]
[RS_LT_00059]	LT shall provide an interface for trace points	[SWS_LOG_20001] [SWS_LOG_20002] [SWS_LOG_20003] [SWS_LOG_20004] [SWS_LOG_20005]
[RS_LT_00060]	LT shall send modeled trace messages	[SWS_LOG_20001] [SWS_LOG_20002] [SWS_LOG_20003] [SWS_LOG_20004] [SWS_LOG_20005]
[RS_LT_00061]	Tracing shall be configurable at compile time	[SWS_LOG_20001] [SWS_LOG_20002] [SWS_LOG_20003] [SWS_LOG_20004] [SWS_LOG_20005]
[RS_LT_00062]	LT shall be configurable to use ARTI to process the tracing information	[SWS_LOG_20001] [SWS_LOG_20002] [SWS_LOG_20003] [SWS_LOG_20004] [SWS_LOG_20005]

Table 6.1: RequirementsTracing

7 Functional specification

This specification defines the usage of the defined C++ Logging API for the Log and Trace. Adaptive Applications can use these functions to forward Log messages to various sinks, for example the network, a serial bus, the console or the file system.

The following functionalities are provided:

1) Methods for initializing the Logging framework (see 7.2.6)
2) Utility methods to convert decimal values into hexadecimal or binary values (see 7.3.1.2)
3) Automatic timestamping of Log messages (see 7.5)
4) Log and trace network bandwith limitation (see chapter 7.6)

Adaptive Applications and Functional Clusters can startup (see 7.1.1) and shutdown (see 7.1.2) all Functional Clusters with direct ARA interfaces (e.g. the Logging framework), by calling `ara::core::Initialize()` or `ara::core::Deinitialize()`.

7.1 Functional Cluster Lifecyle

7.1.1 Startup

In order to initialize the Logging framework, mandatory information needs to be provided to the Logging framework. These information are extracted from the application execution manifest and the AUTOSAR Meta-Model. The execution manifest parameter `Executable.loggingBehavior` defines if the logging functionality should be initialized. Initialization of the Logging framework (via `ara::core::Initialize`) is mandatory before usage of any `ara::log` API. Failure to do so will result in undefined behavior.

[SWS_LOG_00001] ⌈Log message logged before the Logging framework is able to process them (e.g. daemon communication is not established) shall be queued. The queue size is defined by LogAndTraceInstantiation.queueSize. If this size is exceeded the oldest entries shall be discarded.⌋ *(RS_LT_00003, RS_LT_00052)*

7.1.2 Shutdown

Note: after `ara::core::Deinitialize()` is called, the application has to stop using the logging api. It also means that no futher messages can be logged anymore.

[SWS_LOG_00123]{DRAFT} ⌈When `ara::core::Deinitialize()` is called, the `Logging framework` shall pass all queued messages to configured log sinks.⌋*(RS_LT_00003)*

7.2 Necessary Parameters and Initialization

The concept of identifying the user application:
To be able to distinguish the logs of different application instances within a system (e.g. an ECU or even the whole vehicle), every `Application process`, in that system, has to get a particular ID and a description.

The concept of log contexts:
In order to be able to distinguish the logs from different logical groups within an `Application process`, for every context within an `Application process` a particular ID and a description has to be assigned. Every `Application process` can have an arbitrary number of contexts.

`Machine`-specific configuration settings for the Log and Trace functional cluster are collected in `LogAndTraceInstantiation`. The `Application process`es using the `Logging framework` need to supply the following configuration through the application execution manifest:

- Application ID
- Application description
- The default log level, if not set through the manifest a default predefined value is set
- The log mode
- The log file path, in case of a specific log mode that indicates logging to a file

The `Application process` using the `Logging framework` creates a `Logging instance` per context. The context is defined at creation of the `Logging instance` and the following information should be provided:

- Context ID
- Context description
- The default log level, if not set through the manifest a default predefined value is set

7.2.1 Application ID

The Application ID is an identifier that allows to associate generated logging information with its user application. The Application ID is passed as a string value. Depending on the `Logging framework` actual implementation, i.e. `Logging back-end`,

the length of the Application ID might be limited. To be able to unambiguously associate the received logging information to the origin, it is recommended to assign unique Application IDs within one ECU. There is no need for uniqueness of Application IDs across ECUs as the ECU ID will be the differentiator. The system integrator has the overall responsibility to ensure that each Application process instance has a unique Application ID. By having this value defined in the manifest the integrator is able to perform consistency checks. The applicationId in the DltApplication identifies the application instance and is put as ApplicationId into the log and trace message.

Note:
The Application IDs are unique IDs per Application process, meaning if the same Application process is started multiple times it shall have an own ID per instance.

The length limits of the Application ID apply only to version-1 of the PRS LogAndTraceProtocol.

7.2.1.1 Application Description

Since the length of the Application ID can be quite short, an additional descriptive text can be provided. This description is passed as a string and the maximum length is implementation dependent. The applicationDescription in the DltApplication is an optional setting that allows to describe the applicationId as descriptive text.

7.2.2 Default Log Level

The Log severity level represents the severity of the log messages. Severity levels are defined in chapter 7.3. defaultLogThreshold in the DltLogSink defines the initial log reporting level for the application instance.

Each initiated log message is qualified with such a severity level. The default Log severity level is set through the application configuration per Application process. The Log severity level acts as a reporting filter. Only log messages having a higher or the same severity will be processed by the Logging framework, while the others are ignored.

The default Log severity level is the initially configured log reporting level for a certain Application process, though it can be overriden per context.

The Application process wide log reporting level shall be adjustable during runtime. The realization is an implementation detail of the underlying back-end. E.g. remotely via a Logging client for example DLT Viewer. The same applies for the context reporting level.

The design rationale for providing an initial default Log severity level application wide against having per context default Log severity levels is the following:

- It simplifies the API usage. Otherwise the user will have to define a context default `Log severity level` for each group before using the API.
- The context separation of `Log message`s is possible during runtime.

7.2.3 Log Mode

Depending on the `Logging framework` implementation, the passed logging information can be processed in different ways. The destination (the `Log message` sink) can be the console output, a file on the file system or the communication bus. The system integrator is responsible to populate this information in the machine manifest. A direct API for dynamically changing this value for development purposes is provided. In the AUTOSAR Meta-Model the `category` of `DltLogSink` is equivalent to the log mode described here, for more information see [3]. The `category` of `DltLogSink` defines the destination to which the log messages will be forwarded.

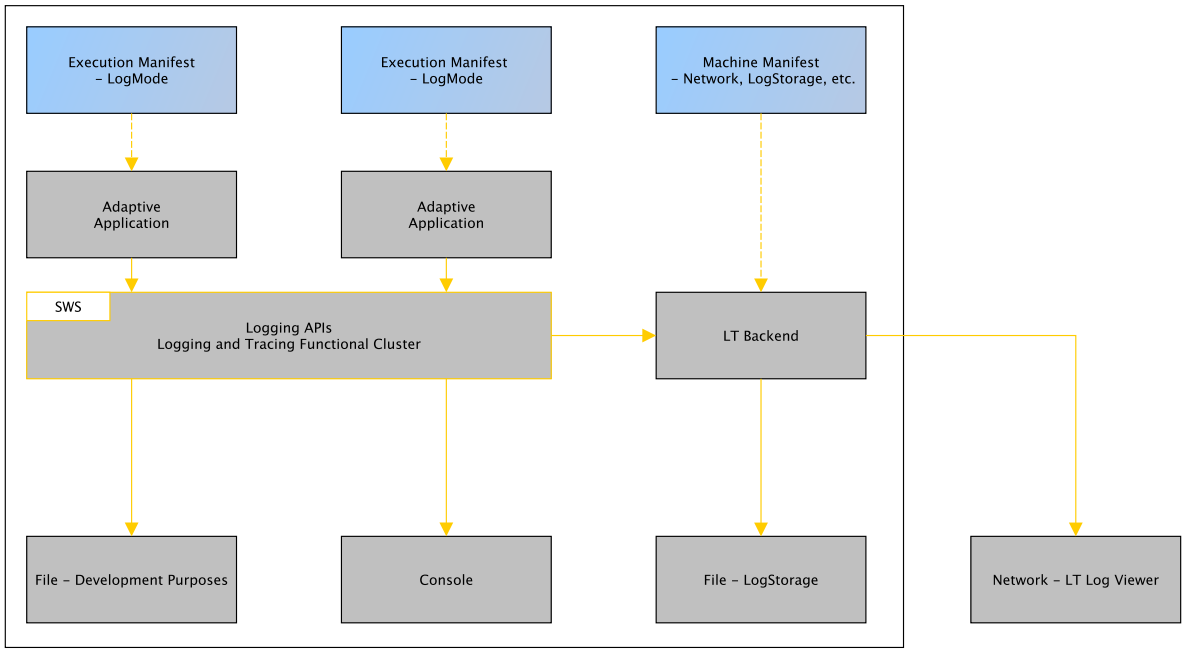

Figure 7.1: Log mode

As shown in the diagram, once the log mode is set to use the `Logging back-end` the configuration is of that back-end is centralized in the `Machine` manifest configuration. For example, the `Logging back-end` can be configured to store the logging information locally and that configuration would be kept in the `Machine`-specific manifest. Furthermore, the output channel on Ethernet for `Log message`s is configured with the `PlatformModuleEthernetEndpointConfiguration` that is referenced by `DltLogSink` in the role `endpointConfiguration`.

7.2.3.1 Log File Path

In case the log mode is set to log to a file, a destination directory path needs to be provided. path in the DltLogSink defines the destination file to which the logging information is passed. This option is provided for development, integration and prototyping purposes and is not suitable for production.

7.2.4 Context ID

The Context ID is an identifier that is used to logically group logging information within the scope of an Application process. The Context ID is passed as a string value. Depending on the actual implementation of the Logging back-end, the length of the Context ID might be limited. Context ID is unique in the scope of an Application process and as such the developer is responsible for assigning it and this information is not modeled in the manifest. There is no need for uniqueness of Context IDs across multiple different Application processes as the Application ID will be the differentiator.

Note:
Special attention should be paid to library components. The libraries are meant to be used by Application processes and therefore are running within the Application process' scope. Logging executed from those libraries will end up inside the scope of the parent Application process. In order to distinguish the internal library logs from the Application process logs or from other library logs within same process, each library might need to reserve its own Context IDs system wide – at least when it shall be used by more than one Application process.

The length limits of the Cotext ID apply only to version-1 of the PRS LogAndTraceProtocol.

7.2.5 Context Description

Since the length of the Context ID can be quite short, an additional descriptive text must be provided. This Context description is passed as a string. The maximum length of the Context description is implementation dependent.

7.2.6 Initialization of the Logging framework

The Application ID and description are used to identify and to associate the provided logging information with the exact process. The log mode and sink information defines where the logging information is routed. Possible destinations are the console, the file system or the communication bus.

From the Application process' perspective, the Logging framework is intialized and a logger instance is created when an Application process decides to register a logging context. These contexts are used to logically cluster logging information.

[SWS_LOG_00002]{DRAFT} ⌈In case of any errors occurring inside the Logging framework or underlying system, it is intended to not bother the Application process and silently discard the function calls. For this purpose, the relevant interfaces neither specify return values nor throw exceptions.⌋*(RS_LT_00003)*

[SWS_LOG_00004]{DRAFT} ⌈The Logging framework shall be initialized if the Executable has a PortPrototype that is typed by a LogAndTraceInterface.⌋*(RS_LT_00003, RS_LT_00047, RS_LT_00048)*

[SWS_LOG_00005] ⌈The function ara::log::CreateLogger shall create a ara::log::Logger instance internally inside the Logging framework and return it as reference to the using application. Before a Log message can be processed, at least one ara::log::Logger shall be available.⌋*(RS_LT_00003, RS_LT_00050)*

Note:
This strong ownership relationship of contexts to the Logging framework ensure correct housekeeping of the involved resources. The design rationale is, once a context is registered against the Logging back-end, its lifetime must be ensured until the end of the Application process.

[SWS_LOG_00006] ⌈The call to ara::log::CreateLogger shall create a ara::log::Logger instance with the given context ID, context description, and log level threshold.⌋*(RS_LT_00003, RS_LT_00050)*

[SWS_LOG_00253] ⌈The call to ara::log::CreateLogger shall create a ara::log::Logger instance with the given context ID and context description, and a log level threshold that is taken from DltLogSink.defaultLogThreshold in the Manifest for the DltLogSink that corresponds to the given context ID. If no such entry exists in the Manifest, then LogLevel::kWarn shall be used.⌋*(RS_LT_00003, RS_LT_00050)*

[SWS_LOG_00254] ⌈The call to ara::log::Logger::SetThreshold shall set the loglevel threshold for the ara::log::Logger instance to the given level.⌋*(RS_LT_00003, RS_LT_00050)*

[SWS_LOG_00007] ⌈Application processes should be able to check if a desired Log severity level is configured through the function ara::log::Logger::IsEnabled. This mechanism conserves CPU and memory resources that are used during preparation of logging information, as this logging information is filtered by the Logging framework later on.⌋*(RS_LT_00003, RS_LT_00045)*

7.3 Log Messages

Log messages can generally be output to different targets. The Log and Trace Functional Cluster supports these logging targets:

- the console
- a file on a local file system
- a network

Most of the discussion in this section assumes that messages are being output to a network, as this use case requires the additional consideration of minimizing network load.

The Log And Trace Functional Cluster offers two principal “classes” of log messages: *Modeled* and *Non-Modeled* messages. Both these support adding one or more “arguments” to a log message. A log message without any arguments serves no purpose and is discarded.

Non-Modeled messages are the traditional way of composing log messages: All arguments of the message are added to an internal message buffer and then eventually serialized for output, either to a console/file, or via network. All parts of the messages will be sent via network. In the DLT protocol, these messages are called “verbose” messages.

Modeled messages are designed to reduce traffic on the network, by omitting certain static (i.e. unchanging) parts of a message from the network. As the name suggests, these parts are instead added to the application ARXML model. In the DLT protocol, these messages are called “non-verbose” messages. A log message viewer application is able to display the full message by combining the static parts from the model with the dynamic parts from the received message.

Non-modeled messages are mainly used during development, as the information required for the modeled messages may not be available at that time. However, non-modeled messages can impose a high load on the network, making modeled messages usually the preferred choice in production systems.

The `ara::log` Functional Cluster supports defining and using both modeled and non-modeled messages in a single application at the same time.

Exception safety: All `Log*()` interfaces are designed to guarantee no-throw behavior. This applies for the whole `Logging API`.

7.3.1 Non-modeled messages

The `ara::log` Functional Cluster defines a "Builder"-pattern inspired set of APIs for constructing non-modeled messages. The `ara::log::Logger::WithLevel` member function is used for creating a `ara::log::LogStream` object which is then subsequently filled with message content (i.e. message arguments). Alternatively to `ara::log::Logger::WithLevel`, there are also separate member functions for creating a `ara::log::LogStream` object, one per supported log level.

Arguments are added to a verbose message by calling an appropriate `operator<<` overload for the desired argument:

```
logger.WithLevel(LogLevel::kInfo) << "text" << 4.2;
```

The `ara::log` Functional Cluster defines such `operator<<` overloads for all C++ arithmetic types, for bool, for string types, and for a number of `ara::core` types. Application-defined data types can be logged as well, by providing suitable `operator<<` overloads for them.

Please note that there is no `operator<<` overload for char, as it might conflict with the overloads for `std::uint8_t` or `std::int8_t`. The behavior of logging a single char is therefore implementation-defined and will often, but not necessarily, output the ordinal value of the char as an integral value.

As a workaround, the `operator<<` overload for `ara::core::StringView` can be used in order to log a single-char string.

As the application model allows "annotating" arguments with attributes, the `ara::log` API for non-modeled messages also supports this. Arguments of certain types can be annotated with a "name" and possibly also a "unit" attribute. For instance:

```
1 logger.WithLevel(LogLevel::kInfo)
2     << Arg("text", "identifier")
3     << Arg(4.2, "velocity", "m/s");
```

The string argument "text" is annotated with a "name" attribute called "identifier". The double argument 4.2 is annotated with a "name" attribute "velocity" and a "unit" of "m/s". These attributes can only be set for some of the built-in types that the `ara::log` API supports, i.e. all arithmetic types, bool, strings, and raw data blobs.

Non-modeled messages can also contain information about the location of the log message call in source code. For this purpose, the member function `ara::log::LogStream::WithLocation` is called with the filename and line number of the call site. These should usually come from the compiler-defined `__FILE__` and `__LINE__` symbols:

```
1 logger
2     .WithLevel(LogLevel::kInfo)
3     .WithLocation(__FILE__, __LINE__) << ...;
```

These are easiest set via a macro-based frontend for `ara::log`, but no such macro has yet been defined in the Adaptive Platform.

[SWS_LOG_00008]{DRAFT} ⌈To initiate a Log message with the Log level Fatal, the API ara::log::Logger::LogFatal shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00009]{DRAFT} ⌈To initiate a Log message with the Log level Error, the API ara::log::Logger::LogError shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00010]{DRAFT} ⌈To initiate a Log message with the Log level Warning, the API ara::log::Logger::LogWarn shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00011]{DRAFT} ⌈To initiate a Log message with the Log level Info, the API ara::log::Logger::LogInfo shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00012]{DRAFT} ⌈To initiate a Log message with the Log level Debug, the API ara::log::Logger::LogDebug shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00013]{DRAFT} ⌈To initiate a Log message with the Log level Verbose, the API ara::log::Logger::LogVerbose shall be called. This API returns a ara::log::LogStream object that has to be used by passing arguments via the insert stream operator<<.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00130]{DRAFT} ⌈To write a Log message with a programmatically determined log level, the API ara::log::Logger::WithLevel shall be called.⌋*(RS_LT_00003, RS_LT_00049)*

Store LogStream objects in a variable:
It is also possible to use the Logging API in an alternative way by storing a ara::log::LogStream object locally in some named variable. The difference to the temporary object is that it won't go out of scope already at the end of the statement, but stays valid and re-usable as long as the variable exists. Hence, it can be fed with data distributed over multiple lines of code. To get the message buffer processed by the Logging framework, the ara::log::LogStream::Flush method needs to be called, otherwise the buffer will be processed when the object dies, i.e. when the variable goes out of scope, at the end of the function block.

Performance remark:
Due to the fact that a ara::log::LogStream is no longer created per message but rather could be re-used for multiple messages, the costs for this object creation is paid only once – per log level. How much this really influences the actual performance depends on the Logging framework implementation. However the main goal of this alternative usage of the Logging API is to get the multi-line builder functionality.

Note:
It is highly advised NOT to hold global `ara::log::LogStream` objects in multi-threaded `Application`s, because then concurrent access protection will no longer be covered by the `Logging API`.

Usage examples:

```
Logger& ctx0 = CreateLogger("CTX0", "Context Description CTX0");
ctx0.LogInfo() << "Some log information" << 123;
```

```
// Locally stored LogStream object will process the arguments
// until either Flush() is called or it goes out of scope from
// the block is was created
Logger& ctx1 = CreateLogger("CTX1", "Context Description CTX1");
LogStream localLogInfo = ctx1.LogInfo();
localLogInfo << "Some log information" << 123;
localLogInfo << "Some other information";
localLogInfo.Flush();
localLogInfo << "a new message..." << 456;
```

7.3.1.1 Message Tags

Arbitrary "message tags" can be set by calling the member function `ara::log::LogStream::WithTag` with an argument that is convertible to `ara::core::StringView`.

Multiple such tags can be added to a message; these tags will then be added to the message in the order of addition.

Example:

```
logger.WithLevel(LogLevel::kInfo)
      .WithTag("filter1")
      .WithTag("filter2")
      << ...;
```

The length of a tag shall not exceed 255 characters. Only characters from the ASCII character set are allowed.

7.3.1.2 Conversion Functions

Sometimes it makes sense to represent integer numbers in hexadecimal or binary format instead of decimal format. Similarly, floating-point numbers often should appear with a certain precision.

For this purpose, it is possible to “annotate” numerical arguments with formatting hints. Viewer applications can use these hints in order to influence the appearance of the received numbers.

Formatting hints can be added with the generic `ara::log::Arg` decorator function, for instance:

```
1 mLog.LogInfo() << Arg(42, Hex());
2 mLog.LogInfo() << Arg(43, Hex(4));
```

uses `ara::log::Hex` to add the "hint" to the message that this argument is supposed to be displayed as a hexadecimal value. The second line additionally adds the hint that the hexadecimal number should be displayed with four digits.

Other functions exist that allow to format as binary, decimal, or octal values, and to format floating-point values in different ways and with specific precisions.

It is the decision of the viewer application whether to heed these hints.

7.3.2 Modelled messages

The `ara::log` Functional Cluster defines a single member function `ara::log::Logger::Log` for sending modeled messages. Unlike the non-modeled message APIs, it represents a single-call interface, i.e. a single function call passes all arguments to the `ara::log::Logger` instance and performs all necessary actions to generate and send the message.

This has the advantage that the runtime cost for a modeled message that is eventually not being output (because the message's log level does not reach the configured log level threshold) can be made very small: after parameter passing and function call, a single `if` clause checks the log level threshold and immediately returns if the threshold is not reached. This contrasts with the non-modeled message APIs, where multiple function calls are performed for constructing a message object, even if that is then eventually discarded.

[SWS_LOG_00240]{DRAFT} ⌈If the API `ara::log::Logger::Log` is called, the implementation shall use the `DltMessage` that corresponds to the parameter `MsgId`.⌋ *(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00241]{DRAFT} ⌈The given argument types shall be verified during compile time and a compile error shall be raised if they do not match to the modeled message argument types .⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00242]{DRAFT} ⌈If the configuration item `DltLogSink.nonVerboseMode` is false, all (modeled) `DltMessage`s sent to this `DltLogSink` shall be composed as verbose messages. Otherwise, non-verbose messages shall be composed.⌋*(RS_LT_00003, RS_LT_00049)*

7.3.2.1 Log message model

All modeled messages are defined as `DltMessage`s, which are aggregated by a `LogAndTraceMessageCollectionSet`. Each `DltMessage` contains a `messageId`,

which needs to be unique within an ECU, and the messageTypeInfo denoting the log level, and optionally the messageSourceFile and messageLineNumber. The DltMessage aggregates an ordered list of DltArguments, which in turn aggregate SwDataDefProps in the role networkRepresentation. The name of a log message argument is taken from the shortName of the DltArgument, while the type and unit are taken from the SwDataDefProps.

At design time it is possible to define an Executable with a SwComponentPrototype that has a PortPrototype typed by an LogAndTraceInterface. Such a PortPrototype is used to describe that the SwComponentPrototype is able to forward logging information onto the external Dlt Log Viewer. The DltMessages that are used by the SwComponentPrototype are collected in the LogAndTraceMessageCollectionSet that is referenced from the PortPrototype.

At deployment time a mapping of the PortPrototype typed by a LogAndTraceInterface to a DltLogSink is described with the DltLogSinkToPortPrototypeMapping that takes a Process of the Executable as context into account.

The DltLogSink is referenced by the LogAndTraceInstantiation that in turn is associated with the DltEcu that carries the ecuId. The LogAndTraceInstantiation itself defines with sessionIdSupport whether session IDs are used or not. The DltLogSinkToPortPrototypeMapping defines the dltSessionId and contains a reference to DltContext that defines the contextId and the corresponding contextDescription. The Process in which the Executable is executed is assigned with the DltApplication that defines the applicationId and the corresponding applicationDescription.

The DltLogSink defines with the category the type of the output sink: DltLogSinkRemote, DltLogSinkDlt, DltLogSinkFile, DltLogSinkConsole.

In case the DltLogSink has the category DltLogSinkRemote to provide means to forward logging information to a Dlt log channel the logChannelId is defined. In case the DltLogSink has the category DltLogSinkDlt to provide means to forward logging information onto on a specific VLAN the endpointConfiguration is defined in addition to logChannelId. The nonVerboseMode describes whether the modeled messages will be sent as verbose messages as if they were non-modeled messages. In case the DltLogSink has the category DltLogSinkFile to provide means to forward logging information to a file in path of a file system a path is defined. In case the DltLogSink has the category DltLogSinkConsole to provide means to forward logging information the console the bufferOutput attribute defines whether a buffer is used or not.

In case DltLogSink.segmentationSupported is enabled, the messages shall use the segmentation information to segment the log messages when they are larger then the MTU of the Ethernet connection.

To create a ara::log::Logger, the application has to call ara::log::CreateLogger with the ara::core::InstanceSpecifier derived from the manifest (a

shortName path from the Executable to a PortPrototype or a mapping derived from FunctionalClusterInteractsWithFunctionalClusterMapping).

[SWS_LOG_00251]{DRAFT} ⌈When ara::log::CreateLogger is called with ara::core::InstanceSpecifier as a parameter, the CtxID, AppID, Ecu ID and session ID shall be taken from the model and connections to logging sinks shall be created for the returned ara::log::Logger object. The InstanceSpecifier presents a PortPrototype typed by LogAndTraceInterface or ServiceInterface according to constr_5286, constr_5287.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00257]{DRAFT} ⌈If the ara::log::Logger::Log function is called on an instance of ara::log::Logger that has no connection to the model, the call shall be ignored.⌋*(RS_LT_00003, RS_LT_00049)*

Note: Such an unconnected ara::log::Logger instance can be obtained, for example, from a call to ara::log::CreateLogger

7.3.2.2 ara::log Modeled Message Generation Principle

This chapter specifies those C++ header files used directly for modeled messages in an Adaptive Application. As part of the Adaptive Platform Methodology, these C++ header files are generated by a workflow tool (ara::log Modeled Message Generator) directly from the LogAndTraceMessageCollectionSet and DltLogSink ARXML configuration as the implementation of type and variable definitions representing modeled messages.

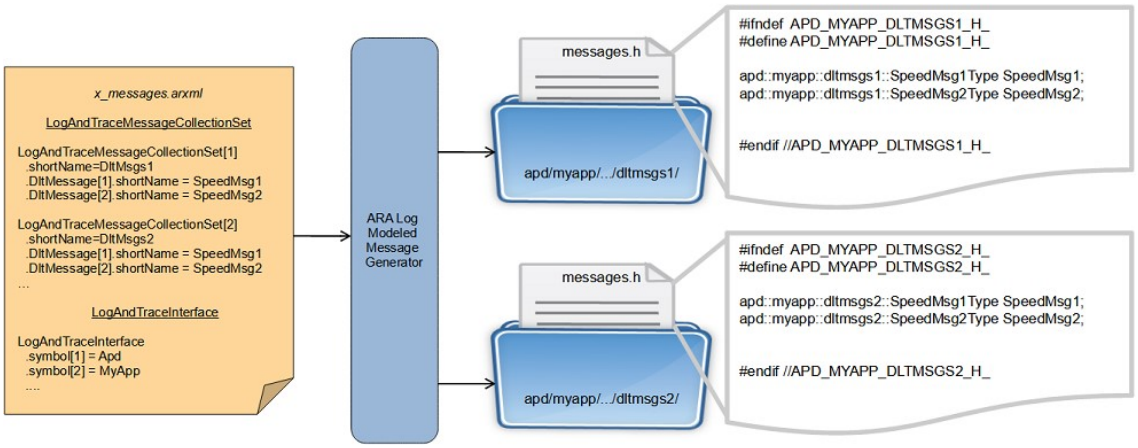

Figure 7.2: ara::log Modeled Message Generator

7.3.2.2.1 Header file directory structure

[SWS_LOG_00244]{DRAFT} **Modeled message header files directory structure** ⌈The modeled message header files defined by [SWS_LOG_00245] shall be located within the folder

```
<namespace[0]>/<namespace[1]>/.../<namespace[n]>/<name>/
```

where:

```
<namespace[0]> ... <namespace[n]>
```

are the namespace names as defined in `LogAndTraceInterface`, converted to lower case, and `<name>` is `LogAndTraceMessageCollectionSet`.shortName, converted to lower case.

⌋*(RS_LT_00003, RS_LT_00049)*

7.3.2.2.2 Modeled message header file

[SWS_LOG_00245]{DRAFT} **Modeled Message header file name** ⌈The modeled message header file shall use the file name `messages.h`.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00246]{DRAFT} **Modeled Message header file existence** ⌈One `LogAndTraceMessageCollectionSet` shall correspond with one modeled message header file.⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00247]{DRAFT} **Inclusion of modeled messages** ⌈The symbols defining the modeled messages of one `LogAndTraceMessageCollectionSet` shall be made exclusively available in the header file defined in [SWS_LOG_00245], [SWS_LOG_00246]⌋*(RS_LT_00003, RS_LT_00049)*

Namespaces are used to separate the symbols defined based on `LogAndTraceMessageCollectionSet` from each other to prevent name conflicts and they allow to use reasonably short names.

7.3.2.2.3 Contents in the modeled message header file

The modeled message header file contains constexpr variable definition and corresponding type definition for the parameters of the API `ara::log::Logger::Log`, and those definitions are defined inside user-defined namespaces to avoid variable or type name collision.

[SWS_LOG_00248]{DRAFT} **Namespaces in the modeled message header files** ⌈Based on the symbol attributes of the ordered `SymbolProps` aggregated by `LogAndTraceInterface` and `LogAndTraceMessageCollectionSet.shortName` in role namespace, the C++ namespace of the modeled message header file shall be:

```
1  namespace <LogAndTraceInterface.namespace[0].symbol> {
2  namespace <LogAndTraceInterface.namespace[1].symbol> {
3  namespace <...> {
4  namespace <LogAndTraceInterface.namespace[n].symbol>{
5  namespace <LogAndTraceMessageCollectionSet.shortName>{
6  ...
7  } // namespace <LogAndTraceMessageCollectionSet.shortName>
8  } // namespace <LogAndTraceInterface.namespace[n].symbol>
9  } // namespace <...>
10 } // namespace <LogAndTraceInterface.namespace[1].symbol>
11 } // namespace <LogAndTraceInterface.namespace[0].symbol>
```

with all namespace names converted to lower-case letters.⌋*(RS_LT_00003, RS_LT_00049)*

Note:
In order to avoid name collisions between types and variables of different `LogAndTraceMessageCollectionSet` in situation where the different `LogAndTraceMessageCollectionSet` carry the same `shortName`, it is highly recommened to place

different LogAndTraceMessageCollectionSet into dedicated unique C++ namespaces. This is achieved by attaching corresponding ordered SymbolProps to the LogAndTraceMessageCollectionSet where the ordered SymbolProps differ in at least one of their symbol attributes.

[SWS_LOG_00249]{DRAFT} **Modeled message header files multiple inclusion guard** ⌈The ara::log Modeled Message Generator shall generate a multiple inclusion guard around the whole header file in each modeled message header file according to the format:

```
#ifndef <path>_H_
#define <path>_H_
 ...
#endif // <path>_H_
```

where <path> is the relative path of the header file according to [SWS_LOG_00244] up to but omitting the file extension, with all path components separated by an underscore ("_"), converted to upper-case letters.

⌋*(RS_LT_00003, RS_LT_00049)*

[SWS_LOG_00250]{DRAFT} **Implementation of DltMessage** ⌈For each DltMessage, the modeled message header file shall contain the following DltMessage variable definition in the same namespace:

constexpr static <DltMessageName>Type <DltMessageName>;
where:
<DltMessageName> is DltMessage.shortName.
<DltMessageName>Type is an implementation-defined data type for DltMessage.⌋
(RS_LT_00003, RS_LT_00049)

7.3.2.3 Usage

To use modeled messages, an ARXML file can be used for generating header files which include constexpr variables within corresponding namespaces.

For instance, if the ARXML representation of the manifest contains the following:

```
<AR-PACKAGES>
<AR-PACKAGE>
  <SHORT-NAME>AUTOSAR</SHORT-NAME>
  <ELEMENTS>
    <LOG-AND-TRACE-INTERFACE>
      <SHORT-NAME>LogAndTraceInterface</SHORT-NAME>
      <NAMESPACES>
        <SYMBOL-PROPS>
          <SHORT-NAME>apd</SHORT-NAME>
          <SYMBOL>apd</SYMBOL>
        </SYMBOL-PROPS>
        <SYMBOL-PROPS>
          <SHORT-NAME>myapp</SHORT-NAME>
          <SYMBOL>myapp</SYMBOL>
```

```
        </SYMBOL-PROPS>
      </NAMESPACES>
    </LOG-AND-TRACE-INTERFACE>
    <ADAPTIVE-APPLICATION-SW-COMPONENT-TYPE>
      <SHORT-NAME>DltMessage_SWC</SHORT-NAME>
      <PORTS>
        <R-PORT-PROTOTYPE>
          <SHORT-NAME>DltMessages_RPort</SHORT-NAME>
          <REQUIRED-INTERFACE-TREF DEST="LOG-AND-TRACE-INTERFACE">/AUTOSAR/
              LogAndTraceInterface</REQUIRED-INTERFACE-TREF>
        </R-PORT-PROTOTYPE>
      </PORTS>
    </ADAPTIVE-APPLICATION-SW-COMPONENT-TYPE>
    <LT-MESSAGE-COLLECTION-TO-PORT-PROTOTYPE-MAPPING>
      <SHORT-NAME>DltMessages_Mapping</SHORT-NAME>
      <LOG-AND-TRACE-MESSAGE-COLLECTION-SET-REF DEST="LOG-AND-TRACE-MESSAGE
          -COLLECTION-SET">/AUTOSAR/DltMessages</LOG-AND-TRACE-MESSAGE-
          COLLECTION-SET-REF>
      <R-PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">/AUTOSAR/DltMessage_SWC
          /DltMessages_RPort</R-PORT-PROTOTYPE-REF>
    </LT-MESSAGE-COLLECTION-TO-PORT-PROTOTYPE-MAPPING>
    <LOG-AND-TRACE-MESSAGE-COLLECTION-SET>
      <SHORT-NAME>DltMessages</SHORT-NAME>
      <DLT-MESSAGES>
        <DLT-MESSAGE>
          <SHORT-NAME>SpeedMsg</SHORT-NAME>
          <DLT-ARGUMENTS>
            <DLT-ARGUMENT>
              <SHORT-NAME>Velocity</SHORT-NAME>
              <LENGTH>1</LENGTH>
              <NETWORK-REPRESENTATION>
                <SW-DATA-DEF-PROPS-VARIANTS>
                  <SW-DATA-DEF-PROPS-CONDITIONAL>
                    <BASE-TYPE-REF DEST="SW-BASE-TYPE">/AUTOSAR/StdTypes/
                        double</BASE-TYPE-REF>
                    <UNIT-REF DEST="UNIT">/AUTOSAR/PhysicalUnits/
                        Units_Blueprint/NoUnit</UNIT-REF>
                  </SW-DATA-DEF-PROPS-CONDITIONAL>
                </SW-DATA-DEF-PROPS-VARIANTS>
              </NETWORK-REPRESENTATION>
            </DLT-ARGUMENT>
          </DLT-ARGUMENTS>
          <MESSAGE-LINE-NUMBER>0</MESSAGE-LINE-NUMBER>
          <MESSAGE-SOURCE-FILE>source_file_path</MESSAGE-SOURCE-FILE>
          <MESSAGE-TYPE-INFO>DLT_TRACE_STATE</MESSAGE-TYPE-INFO>
        </DLT-MESSAGE>
      </DLT-MESSAGES>
    </LOG-AND-TRACE-MESSAGE-COLLECTION-SET>
  </ELEMENTS>
</AR-PACKAGE>
</AR-PACKAGES>
```

and the source code contains this code sequence:

```
#include "ara/log/logger.h"
#include "apd/myapp/dltmessages/messages.h"
```

```
3
4  double value= 2.3;
5  Logger& logger = ...
6  logger.Log(apd::myapp::dltmessages::SpeedMsg, value);
```

then the `DltMessage` definition will be generated in a given namespace, `apd::myapp::dltmessages`. For each `LogAndTraceMessageCollectionSet`, an own header file will be generated, like apd/myapp/dltmessages/messages.h. The framework will define a global `constexpr` variable called `SpeedMsg` within the namespace `apd::myapp::dltmessages` in the header file. This variable contains knowledge about the message's modeled aspects, such as parameter types and log level, allowing the `ara::log` implementation to verify that the number and types of parameters given to `ara::log::Logger::Log` matches the model of the particular message.

For using modeled messages, it is necessary to explicitly include the header file for the corresponding `LogAndTraceMessageCollectionSet`. In the example above, it is apd/myapp/dltmessages/messages.h.

7.3.2.4 Customizing message properties

Certain properties can be customized for an individual modeled message, by using the separate API `ara::log::Logger::LogWith` that allows to add attributes to a modeled message. For instance:

```
1 logger.LogWith(std::make_tuple(Location(FILE, LINE)), SpeedMsg, 4.2);
```

Supported attributes are:

Location: file identifier and line number of the location where the log message is being sent, similar to `ara::log::LogStream::WithLocation`

Tag: Arbitrary message tag, similar to `ara::log::LogStream::WithTag`

7.4 Segmentation

L&T module decides based on the knowledge of the lower layer frame length, whether segmentation needs to be used. (case: usage of segmentation for shorter frames needs not to be considered). Overall length of segmented messages is limited to $< 2^{48}$ bytes (i.e. 16bit header length * 32bit sequence).

Proposal for the segementation header: (no handshake needed)

- FrameType (8bit): FirstFrame, ConsecutiveFrame, LastFrame, AbortFrame;
- TotalLength (64bit), only with FirstFrame;
- SequenceCounter (32bit), only with ConsecutiveFrame;

- - AbortReason (8bit), only with AbortFrame;

LastFrame indicates successful transmission of the whole message. Successful reception can be assumed if the accumulated length of the data contained in the segmented frames equals the TotalLength of the FirstFrame. In the BaseHeader the HeaderType (HTYP2).Bit 11 shall be reservered for the SegmentationHeader (WS = WithSegmentation). In the ExtensionHeader the length field of the SegmentationHeader will depend on the FrameType;

7.5 Log and Trace Timestamp

The Log and Trace information is transmitted by means of the LT protocol which is bus agnostic.
This protocol offers the possibility to include a timestamp in each sent message, as long as such messages are sent with an extended header (refer to [7] for more information).

The synchronized time base is supplied by the Time Synchronization Functional Cluster. The now() method is used by the Adaptive Applications in order to retrieve the current time from the TS (refer to [8] for more information).

According to the requirement [TPS_MANI_03162], the reference time base is derived from the machine manifest timeBaseResource.

[SWS_LOG_00082] ⌈Log and Trace should have accesss to a synchronized time base. The attribute timeBaseResource in LogAndTraceInstantiation shall be used to identify the time base.⌋*(RS_LT_00003)*

[SWS_LOG_00083] ⌈If the time base resource is referenced by the Log and Trace module in the manifest configuration LogAndTraceInstantiation.timeBaseResource, the current timestamp information shall be included in each DLT message, otherwise not.⌋*(RS_LT_00003, RS_LT_00017)*

[SWS_LOG_00091]{DRAFT} ⌈When the CreateLogger() function is called, Log and Trace shall send a message, "local time base used" in case the used time base is a local time base or "global time base used" in case the used time base is a globally synchronized time base.

⌋*(RS_LT_00003)*

7.6 Log and Trace data loss prevention

[SWS_LOG_00095]{DRAFT} ⌈When `Log and Trace` receives simultaneously a high load of trace information generated by multiple `Adaptive Application`s, it shall buffer this data internally to prevent the data loss during its continuous transmission.⌋ *(RS_LT_00003, RS_LT_00030)*

7.7 Tracing

Tracing is crucial for a successful and effective application development and is needed, for determination of the timing behavior of their Applications. Tracing of the behavior of the Adaptive Platform and of Adaptive Applications has paramount advantage when analyzing information flow for numerous reasons, from debugging to measuring communication latencies to profiling different communication events.

Log & Trace introduces an overhead in the application. This overhead may influence the behavior of the application and uses resources as memory, execution time etc. On the other hand there are different uses cases during development of an application and running the application in the field. Thus sometimes it is necessary to log as many data as possible while the resources don't matter and sometimes the used resources have to be as little as possible. To achieve different goals, the different configurations of the Log & Trace should be considered. These configurations include:

- Precompile configuration,
- Static configuration,
- Dynamic configuration.

The precompile configuration configures the Log & Trace that none or minimal runtime overhead does exist in the application. Dedicated trace points can be enabled or disabled before the building of the application. Even so it is possible to configure trace points to use special trace capabilities of the environment. This guarantees minimal runtime overhead during the tweaking performance of the system. The static configuration is part of the deployed application. Certain configuration items i.e. the Log & Trace sinks that are not changeable when the application is running. Checks or initialization is done once when the application is started. The dynamic configuration introduces the most overhead because the checks or initialization has to be done each time a Log & Trace is called.

The trace methods are integrated in the ara::log Functional Cluster. For AUTOSAR applications the trace is identical to the use of the API for modeled messages. This chapter specifies internal handling of the trace to provide a clear interface between the AUTOSAR stack vendor and the trace tool vendor. It further shows special adaptions and specifies the configuration of the `ARTI-trace`.

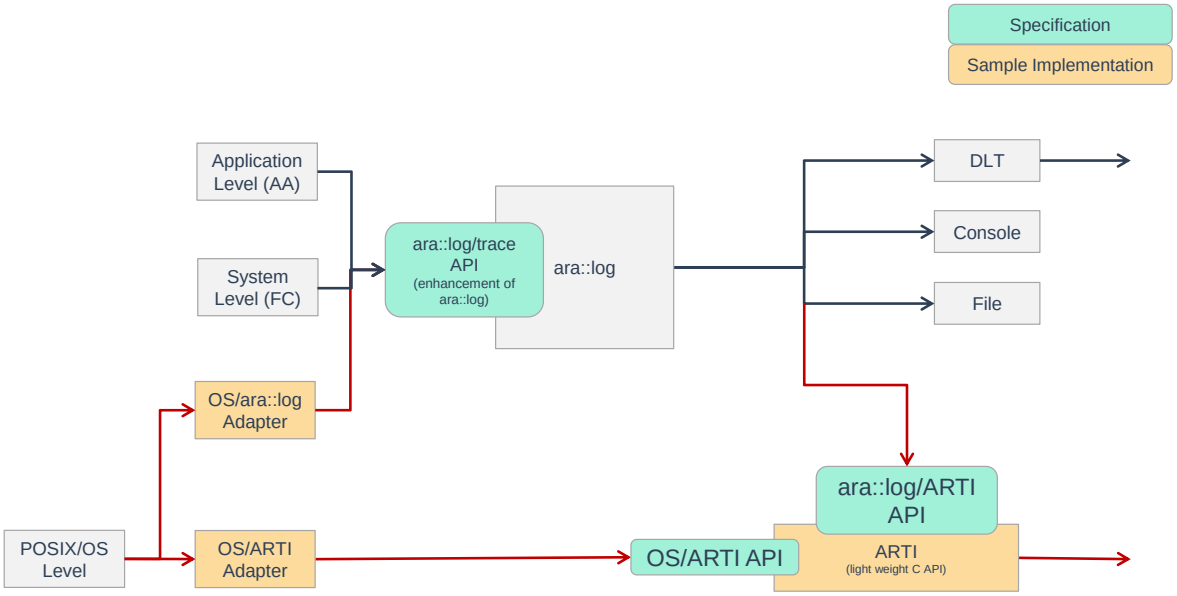

Figure 7.3: Overview of Tracing

7.7.1 OS/ara::log Adapter

The OS/ara::log Adapter adapts the trace functionality of the operating system to ara::log. Typically this adapter may be implemented as an additional daemon that maps kernel trace events to ara::log messages. This consumes runtime resources but provides the full flexibility of ara::log. The OS/ara::log Adapter is specified in [9, SWS OperatingSystemInterface] chapter OS/ara::log Adapter.

7.7.2 Precompile configuration of ara::log/Trace

The ara::log/Trace is configured with precompile configuration to allow only minimal or no overhead in the running application. Tracing with ara::log/Trace uses modeled messages to signal events.

From application side the tracing is identical to logging modeled messages. The modeled messages and its parameters are arguments of the modeled logging API:

```
template< typename MsgId, typename ... Params >
    Log(const MsgId& msg_id, const Params& ...params) noexcept;
```

Based on the `Executable.traceSwitchConfiguration` and the `MsgId` the messages are further provided to the logger or/and to `ara::log::ext::TraceArti`. If only `ara::log::ext::TraceArti` is configured, then this message can be used for performance measurement without the overhead of log. If only log is configured, or if

a message is not configured for tracing, then the message is provided directly to the logger.

[SWS_LOG_20001]{DRAFT} ⌈Modeled messages shall be routed to the logger if `Executable.traceSwitchConfiguration` is configured to `traceSwitchLog` for that message or if there is no entry in `Executable.traceSwitchConfiguration` for that message. The evaluation of routing shall be done at compile time.⌋*(RS_LT_00059, RS_LT_00060, RS_LT_00061, RS_LT_00062)*

[SWS_LOG_20002]{DRAFT} ⌈Modeled messages shall be routed to `ara::log::ext::TraceArti` if `Executable.traceSwitchConfiguration` is configured to `traceSwitchArti` for that message. The evaluation of routing shall be done at compile time.⌋*(RS_LT_00059, RS_LT_00060, RS_LT_00061, RS_LT_00062)*

[SWS_LOG_20003]{DRAFT} ⌈Modeled messages shall be routed to `ara::log::ext::TraceArti` and to the logger if `Executable.traceSwitchConfiguration` is configured to `traceSwitchArtiAndLog` for that message. The evaluation of routing shall be done at compile time.⌋*(RS_LT_00059, RS_LT_00060, RS_LT_00061, RS_LT_00062)*

[SWS_LOG_20004]{DRAFT} ⌈Modeled messages shall be discarded if `Executable.traceSwitchConfiguration` is configured to `traceSwitchNone` for that message. The evaluation of routing shall be done at compile time.⌋*(RS_LT_00059, RS_LT_00060, RS_LT_00061, RS_LT_00062)*

If any message has to be routed the `ara::log::ext::TraceArti`, then the header file `trace_arti_spec.h` has to be included there. `trace_arti_spec.h` contains he generated specialization of `ara::log::ext::TraceArti`. `trace_arti_spec.h` is located in the directory of the generated modeled messages.

The base template function

```
template< typename MsgId, typename ... Params >
    void TraceArti( const MsgId& msg_id, const Params& ...params ) noexcept
{
}
```

is declared in `ara/log/trace_arti.h`.

The spezializations of the function `ara::log::ext::TraceArti` are generated by the code generator of the trace tool vendor. These generated functions are based on the configuration of the trace tool. The generated header file is named `trace_arti_spec.h` and is located where the header files of the modeled messeges are located. Specializations of `ara::log::ext::TraceArti` are used to adapt the API to the capabilities of the used trace tool. For example, the basic trace features of the system like `LTTNG` can also serve as "trace tool".

Example message routing for minimal overhead can be achieved with the decorator pattern and template specialization. Hereby the `ara::log::Logger` decorates the `LoggerImpl`:

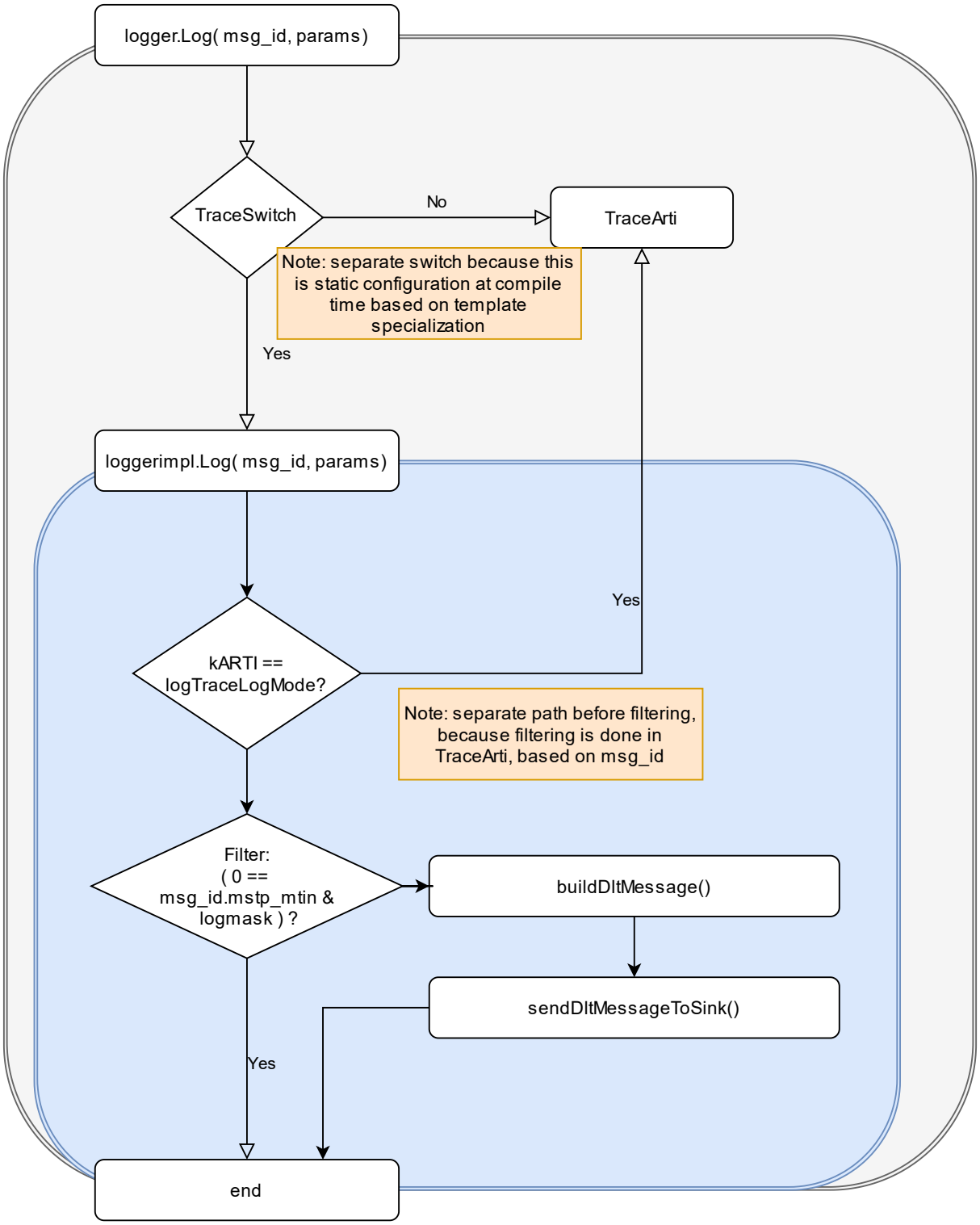

Figure 7.4: Example program flow of ara::log/Trace

The program flow is as follows: The application or functional cluster calls `logger.Log()`. This forwards the message to `TraceSwitch()`, which is a `MsgId` based, compile-time filter that forwards the call to the trace implementation of the trace tool (`ara::log::ext::TraceArti`) and/or to the log implementation

(`LoggerImpl.Log()`), depending on the configuration. The entire switching is based on template specialization and is done at compile time. It does not cost any runtime.

The following example of the `ara::log::Logger` implementation for modeled messages uses `ara::log::ext::TraceArti` to trace to ARTI, and `LoggerImpl` to log to the standard log path. `MsgId` is generated out of the message model.

```
class LoggerImpl // the decorated logger
{
    public:
    template< typename MsgId, typename ... Params >
        void Log(const MsgId& msg_id, const Params& ... params) noexcept
    {
        // .... do the dlt logging here
    }
};

class Logger // the decoration
{
    // all other functions are forwarded to the decorated m_impl
    public:
    template< typename MsgId, typename ... Params >
        void Log(const MsgId& msg_id, const Params& ...params) noexcept
        {
            TraceSwitch( m_impl, msg_id, params ... );
        }
    private:
    LoggerImpl m_impl; // the decoreated class
};
```

The decorated `logger.Log()` calls the calls `TraceSwitch` function template.

```
template< typename MsgId, typename ... Params >
    void TraceSwitch( LoggerImpl& logger, const MsgId& msg_id,
         const Params& ...params ) noexcept;
    {
        // by default (no specialization) the decorated logger is called
        logger.Log( msg_id, params ... );
    }

// specialized with ExecutionManagerProcessStateChangeMsgType
template< typename ... Params >
    void TraceSwitch( LoggerImpl& logger,
                      const ExecutionManagerProcessStateChangeMsgType&
    msg_id,
                      const Params& ...params ) noexcept
    {
        TraceArti( msg_id, params... );
    }
```

In case of `ExecutionManagerProcessStateChangeMsgType` the `TraceSwitch` calls the specified `ara::log::ext::TraceArti`.

`TraceSwitch` is used to configure the routing of the message at compile time, based on its type. This allows to configure the routing for each message separately, which gives the possibility to route messages based on use case, e.g. performance measurement.

The TraceSwitch is generated by the code generator of the AUTOSAR-Stack vendor. The generated header file should be located in the same directory where the modeled messages are located.

7.7.3 Static configuration of ara::log/Trace

Modeled ara::log messages can be routed to ARTI trace as additional log sink beside console, file and network. This allows the trace to be configured statically. The ARTI trace can be switched on and off without recompiling the application.

[SWS_LOG_20005]{DRAFT} ⌈When DltLogChannel.logTraceLogMode is set to kArti, then the ara::log::Logger::Log shall call ara::log::ext::TraceArti with all variadic parameters.⌋*(RS_LT_00059, RS_LT_00060, RS_LT_00061, RS_LT_00062)*

The call of ara::log::ext::TraceArti should be done before the message is being formated or filtered by the verbosity because the filtering for ARTI is done by the generated ara::log::ext::TraceArti or by the trace tool itself.

7.8 Output Bindings

7.8.1 Overview

Log and Trace has built-in support for console output and file output. In addition to that, it supports a “remote” backend, which represents a network-based sink. A typical choice on an AUTOSAR Machine is the LT protocol, but other backends can also be supported by vendors.

This section defines the mapping of the abstract ara::log API to such backends.

7.8.2 Console binding

7.8.2.1 General

The console output binding is set by using the value DLT_LOGSINK_CONSOLE for a DltLogSink referenced by LogAndTraceInstantiation.

Please note that many aspects of the console output are currently implementation-defined. This includes, for instance, the general layout of messages, the format of timestamps, and which elements from the DLT message header are being output at all.

[SWS_LOG_00227] Newline addition in console output ⌈The Logging framework shall append an EOL sequence to each message in console output.⌋*(RS_LT_00002)*

7.8.2.2 Message arguments

[SWS_LOG_00228] Argument separation in console output ⌈When a message contains more than one payload argument, any two payload arguments shall be separated by a single space character (U+0020) in console output.⌋ *(RS_LT_00002)*

Example: A log call such as:
`mLog.LogInfo() << "speed" << 4.1 << "m/s";`
will result in this output:
`... speed␣4.1␣m/s`
where each "␣" represents a single space character.

[SWS_LOG_00229] Display of argument attributes in console output ⌈If a payload argument contains a "name" attribute, then the "name" attribute shall be prepended to the argument value, and separated from the value with a colon character (U+003a). If a payload argument contains a "unit" attribute, then the "unit" attribute shall be appended to the argument value, and separated from the value with a colon character (U+003a) as well.⌋ *(RS_LT_00056, RS_LT_00002)*

Example: `speed:42.1:km/h`, where "speed" is the value of the "name" attribute, and "km/h" is the value of the "unit" attribute

The following subsections details how the data types that are supported by the ara::log API are being represented in console output.

7.8.2.2.1 bool

[SWS_LOG_00230] Console output of bool values ⌈A message argument value of `bool` type shall be output to console as the string "0" (U+0030) for `false`, or "1" (U+0031) for `true`.⌋ *(RS_LT_00002)*

7.8.2.2.2 ara::core::Span<const ara::core::Byte>

[SWS_LOG_00231] Console output of raw binary data ⌈A message argument value of type `ara::core::Span<const ara::core::Byte>` shall be output to console as a sequence of hexadecimal octets separated by apostrophe characters (U+0027).⌋ *(RS_LT_00002)*

Example: `48'65'6c'6f`

7.8.2.2.3 InstanceSpecifier

[SWS_LOG_00232] Console output of InstanceSpecifier values ⌈A message argument value of type `ara::core::InstanceSpecifier` shall be output to console as the result of calling `InstanceSpecifier::ToString`.⌋*(RS_LT_00002)*

7.8.2.2.4 std::chrono::duration

[SWS_LOG_00233] Console output of std::chrono::duration values ⌈A message argument value of type `std::chrono::duration` shall be output to console as a decimal integer value, followed by the duration's unit in SI notation, for at least these unit dimensions: `std::nano`, `std::micro`, `std::milli`, `std::centi`, `std::deci`, `std::ratio<1>`, `std::deca`, `std::hecto`, `std::kilo`, `std::mega`, `std::giga`.⌋*(RS_LT_00002)*
Note: The SI metric prefix for "micro" may be "µ" or simply "u".

Example: `42ns`

7.8.3 File binding

The file output binding is set by using the value `DLT_LOGSINK_FILE` for a `DltLogSink` referenced by `LogAndTraceInstantiation`.

The appearance of the "file" output is currently implementation-defined.

7.8.4 DLT binding

7.8.4.1 General

An output binding for the DLT protocol is set by using the value `DLT_LOGSINK_NETWORK` or `DLT_LOGSINK_DLT` for a `DltLogSink` referenced by `LogAndTraceInstantiation`.

7.8.4.2 Log level

The `LT protocol` supports the exact same log levels as `ara::log` does. There is therefore a 1:1 mapping of `ara::log` log levels to `LT protocol` log levels.

[SWS_LOG_00234] Translation of log levels ⌈A log level attribute of a `DltMessage` shall be translated into the `LT protocol` log level with the same name.⌋*(RS_LT_00002)*

7.8.4.3 Message arguments

The following subsections details how the data types that are supported by the ara::log API are being represented in the `LT protocol`.

7.8.4.3.1 bool

[SWS_LOG_00235] DLT output of bool values ⌈A message argument of `bool` type shall be output as a DLT message payload argument of type BOOL.⌋*(RS_LT_00002)*

7.8.4.3.2 string types

[SWS_LOG_00236] DLT output of string values ⌈A message argument of type `const char*` or `ara::core::StringView` shall be output as a DLT message payload argument of type STRG, with TYFM set to UTF-8 encoding.⌋*(RS_LT_00025, RS_LT_00002)*

7.8.4.3.3 ara::core::Span<const ara::core::Byte>

[SWS_LOG_00237] DLT output of raw binary data ⌈A message argument of type `ara::core::Span<const ara::core::Byte>` shall be output as a DLT message payload argument of type RAWD.⌋*(RS_LT_00002)*

7.8.4.3.4 InstanceSpecifier

[SWS_LOG_00238] DLT output of InstanceSpecifier values ⌈A message argument of type `InstanceSpecifier` shall be output as a DLT message payload argument of type STRG, with TYFM set to UTF-8 encoding, consisting of the result of calling `InstanceSpecifier::ToString`.⌋*(RS_LT_00002)*

8 API specification

8.1 API Common Data Types

8.1.1 LogLevel

[SWS_LOG_00018]{DRAFT} **Definition of API enum ara::log::LogLevel** ⌈

Kind:	enumeration	
Header file:	#include "ara/log/common.h"	
Forwarding header file:	#include "ara/log/log_fwd.h"	
Scope:	namespace ara::log	
Symbol:	LogLevel	
Underlying type:	std::uint8_t	
Syntax:	`enum class LogLevel : std::uint8_t {...};`	
Values:	kOff= 0x00	No logging.
	kFatal= 0x01	Fatal error, not recoverable.
	kError= 0x02	Error with impact to correct functionality.
	kWarn= 0x03	Warning if correct behavior cannot be ensured.
	kInfo= 0x04	Informational, providing high level understanding.
	kDebug= 0x05	Detailed information for programmers.
	kVerbose= 0x06	Extra-verbose debug messages (highest grade of information)
Description:	List of possible severity levels .	

⌋*(RS_LT_00003)*

8.1.2 Format specifier

[SWS_LOG_00206] Definition of API enum ara::log::Fmt ⌈

Kind:	enumeration	
Header file:	#include "ara/log/logger.h"	
Forwarding header file:	#include "ara/log/log_fwd.h"	
Scope:	namespace ara::log	
Symbol:	Fmt	
Underlying type:	std::uint16_t	
Syntax:	`enum class Fmt : std::uint16_t {...};`	
Values:	kDefault= 0	implementation-defined formatting
	kDec= 1	decimal (signed/unsigned)
	kOct= 2	octal
	kHex= 3	hexadecimal
	kBin= 4	binary
	kDecFloat= 5	decimal float (like printf "%f")
	kEngFloat= 6	engineering float (like printf "%e")

▽

△

	kHexFloat= 7	hex float (like printf "%a")
	kAutoFloat= 8	automatic "shortest" float (like printf "%g")
Description:	Format specifiers for log message arguments .	

⌋(*RS_LT_00046*)

[SWS_LOG_00207] Definition of API class ara::log::Format ⌈

Kind:	struct
Header file:	#include "ara/log/logger.h"
Forwarding header file:	#include "ara/log/log_fwd.h"
Scope:	namespace ara::log
Symbol:	Format
Syntax:	`struct Format final {...};`
Description:	A type holding a formatting hint. The interpretation of precision depends on fmt: For integral types (i.e. Fmt::kDec, Fmt::kOct, Fmt::kHex, Fmt::kBin), precision is interpreted as the minimum number of digits to output, similar to e.g. std::printf("%.7d"). For the floating-point specifiers Fmt::kDecFloat, Fmt::kEngFloat and Fmt::kHexFloat, precision denotes the exact number of digits to be shown after the decimal point; for Fmt::kAutoFloat, precision denotes the number of significant digits to be shown according to the rules of the std::printf "%g" specifier. If fmt is Fmt::kDefault, the precision field is ignored, and an implementation-defined formatting is applied. For integral types, if precision is 0, it is interpreted the same as if it was 1.

⌋(*RS_LT_00046*)

[SWS_LOG_00225] Definition of API variable ara::log::Format::fmt ⌈

Kind:	variable
Header file:	#include "ara/log/logger.h"
Scope:	struct ara::log::Format
Symbol:	fmt
Type:	Fmt
Syntax:	`Fmt fmt;`
Description:	the format specifier

⌋(*RS_LT_00046*)

[SWS_LOG_00226] Definition of API variable ara::log::Format::precision ⌈

Kind:	variable
Header file:	#include "ara/log/logger.h"
Scope:	struct ara::log::Format
Symbol:	precision
Type:	std::uint16_t
Syntax:	`std::uint16_t precision;`
Description:	the precision to use

⌋(*RS_LT_00046*)

For convenience, there are helper functions to create specific instances of `struct Format`; these should generally be used instead of creating `Format` instances manually.

[SWS_LOG_00208] Definition of API function ara::log::Dflt ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Dflt()	
Syntax:	`constexpr Format Dflt () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kDefault formatting hint.	

⌋(*RS_LT_00046*)

[SWS_LOG_00211] Definition of API function ara::log::Dec ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Dec()	
Syntax:	`constexpr Format Dec () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kDec formatting hint and default precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00212] Definition of API function ara::log::Dec ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Dec(std::uint16_t precision)	
Syntax:	`constexpr Format Dec (std::uint16_t precision) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kDec formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00213] Definition of API function ara::log::Oct ⌈

Kind:	function
Header file:	#include "ara/log/logger.h"
Scope:	namespace ara::log

▽

△

Symbol:	Oct()	
Syntax:	`constexpr Format Oct () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kOct formatting hint and default precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00214] Definition of API function ara::log::Oct ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Oct(std::uint16_t precision)	
Syntax:	`constexpr Format Oct (std::uint16_t precision) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kOct formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00209] Definition of API function ara::log::Hex ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Hex()	
Syntax:	`constexpr Format Hex () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kHex formatting hint and default precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00210] Definition of API function ara::log::Hex ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Hex(std::uint16_t precision)	
Syntax:	`constexpr Format Hex (std::uint16_t precision) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kHex formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00215] Definition of API function ara::log::Bin ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Bin()	
Syntax:	`constexpr Format Bin () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kBin formatting hint and default precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00216] Definition of API function ara::log::Bin ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Bin(std::uint16_t precision)	
Syntax:	`constexpr Format Bin (std::uint16_t precision) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kBin formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00217] Definition of API function ara::log::DecFloat ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	DecFloat(std::uint16_t precision=6)	
Syntax:	`constexpr Format DecFloat (std::uint16_t precision=6) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kDecFloat formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00218] Definition of API function ara::log::DecFloatMax ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	DecFloatMax()	
Syntax:	`constexpr Format DecFloatMax () noexcept;`	
Return value:	Format	a Format instance

▽

△

Exception Safety:	noexcept
Description:	Create a Format instance with Fmt::kDecFloat formatting hint and a precision that is sufficient for full round-trip safety.

⌋(*RS_LT_00046*)

[SWS_LOG_00219] Definition of API function ara::log::EngFloat ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	EngFloat(std::uint16_t precision=6)	
Syntax:	`constexpr Format EngFloat (std::uint16_t precision=6) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kEngFloat formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00220] Definition of API function ara::log::EngFloatMax ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	EngFloatMax()	
Syntax:	`constexpr Format EngFloatMax () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kEngFloat formatting hint and a precision that is sufficient for full round-trip safety.	

⌋(*RS_LT_00046*)

[SWS_LOG_00221] Definition of API function ara::log::HexFloat ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	HexFloat(std::uint16_t precision)	
Syntax:	`constexpr Format HexFloat (std::uint16_t precision) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kHexFloat formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00222] Definition of API function ara::log::HexFloatMax ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	HexFloatMax()	
Syntax:	`constexpr Format HexFloatMax () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kHexFloat formatting hint and a precision that is sufficient for full round-trip safety.	

⌋(*RS_LT_00046*)

[SWS_LOG_00223] Definition of API function ara::log::AutoFloat ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	AutoFloat(std::uint16_t precision=6)	
Syntax:	`constexpr Format AutoFloat (std::uint16_t precision=6) noexcept;`	
Parameters (in):	precision	the precision to use
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kAutoFloat formatting hint and given precision.	

⌋(*RS_LT_00046*)

[SWS_LOG_00224] Definition of API function ara::log::AutoFloatMax ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	AutoFloatMax()	
Syntax:	`constexpr Format AutoFloatMax () noexcept;`	
Return value:	Format	a Format instance
Exception Safety:	noexcept	
Description:	Create a Format instance with Fmt::kAutoFloat formatting hint and a precision that is sufficient for full round-trip safety.	

⌋(*RS_LT_00046*)

8.1.3 ClientState

[SWS_LOG_00098]{DRAFT} **Definition of API enum ara::log::ClientState** ⌈

Kind:	enumeration	
Header file:	#include "ara/log/common.h"	
Forwarding header file:	#include "ara/log/log_fwd.h"	
Scope:	namespace ara::log	
Symbol:	ClientState	
Underlying type:	std::int8_t	
Syntax:	`enum class ClientState : std::int8_t {...};`	
Values:	kUnknown= -1	DLT back-end not up and running yet, state cannot be determined.
	kNotConnected	No remote client detected.
	kConnected	Remote client is connected.
Description:	Client state representing the connection state of an external client. .	

⌋(*RS_LT_00003*)

8.2 Function definitions

8.2.1 CreateLogger

[SWS_LOG_00021]{DRAFT} **Definition of API function ara::log::CreateLogger** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	CreateLogger(ara::core::StringView ctxId, ara::core::StringView ctxDescription, LogLevel ctxDefLogLevel)	
Syntax:	`Logger & CreateLogger (ara::core::StringView ctxId, ara::core::StringView ctxDescription, LogLevel ctxDefLogLevel) noexcept;`	
Parameters (in):	ctxId	The context ID.
	ctxDescription	The description of the provided context ID.
	ctxDefLogLevel	The default log level, set to Warning severity if not explicitly specified.
Return value:	Logger &	Reference to the internal managed instance of a Logger object. Ownership stays within the Logging framework
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Creates a Logger object, holding the context which is registered in the Logging framework. If no model is available the sink shall be the console per default.	

⌋(*RS_LT_00003*)

[SWS_LOG_00263]{DRAFT} **Definition of API function ara::log::CreateLogger** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	CreateLogger(ara::core::StringView ctxId, ara::core::StringView ctxDescription)	
Syntax:	`Logger & CreateLogger (ara::core::StringView ctxId, ara::core::StringView ctxDescription) noexcept;`	
Parameters (in):	ctxId	The context ID
	ctxDescription	The description of the provided context ID
Return value:	Logger &	reference to the internal managed instance of a Logger object Ownership stays within the Logging framework
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Creates a Logger object, holding the context which is registered in the Logging framework. If no model is available the sink shall be the console per default.	

⌋(*RS_LT_00003*)

[SWS_LOG_00256]{DRAFT} **Definition of API function ara::log::CreateLogger** ⌈

Kind:	function
Header file:	#include "ara/log/logger.h"
Scope:	namespace ara::log

▽

△

Symbol:	CreateLogger(const ara::core::InstanceSpecifier &is)	
Syntax:	`Logger & CreateLogger (const ara::core::InstanceSpecifier &is) noexcept;`	
Parameters (in):	is	an InstanceSpecifier
Return value:	Logger &	reference to the internal managcifiered instance of a Logger object
Exception Safety:	noexcept	
Description:	Creates a Logger object derived from the manifest.	

⌋(*RS_LT_00003*)

8.2.2 RegisterConnectionStateHandler

[SWS_LOG_00205]{DRAFT} **Definition of API function ara::log::RegisterConnectionStateHandler** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	RegisterConnectionStateHandler(std::function< void(ClientState)> callback)	
Syntax:	`void RegisterConnectionStateHandler (std::function< void(ClientState)> callback) noexcept;`	
Parameters (in):	callback	a callback that is invoked whenever the connection state has changed
Return value:	None	
Exception Safety:	noexcept	
Description:	Register a callback to be invoked whenever the connection state changes. Only a single function can be installed this way; if a callback was already registered before, only the newly registered one is used henceforth.	

⌋(*RS_LT_00003*)

8.2.3 Wrapper object creator

[SWS_LOG_00201]{DRAFT} **Definition of API function ara::log::Arg** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	namespace ara::log	
Symbol:	Arg(T &&arg, const char *name=nullptr, const char *unit=nullptr, Format format=Dflt())	
Syntax:	`template <typename T>` `Argument< T > Arg (T &&arg, const char *name=nullptr, const char *unit=nullptr, Format format=Dflt()) noexcept;`	
Parameters (in):	arg	an argument payload object
	name	an optional "name" attribute for arg
	unit	an optional "unit" attribute for arg

▽

△

	format	an optional formatting hint for integral or floating-point arguments; the default is to use the implementation's standard formatting
Return value:	Argument< T >	a wrapper object holding the supplied arguments
Exception Safety:	noexcept	
Description:	Create a wrapper object for the given arguments. Calling this function shall be ill-formed if any of these conditions are met: • T is not an arithmetic type and not "bool" and not convertible to "ara::core::StringView" and not convertible to "ara::core::Span<const ara::core::Byte>" • T is convertible to "ara::core::StringView" or convertible to "ara::core::Span<const ara::core::Byte>" or "bool", and "unit" is not "nullptr"	

⌋(*RS_LT_00003*)

8.3 Class definitions

8.3.1 Class LogStream

The class `ara::log::LogStream` represents a `Log message`, allowing stream operators to be used for appending data.

[SWS_LOG_00173]{DRAFT} **Definition of API class ara::log::LogStream** ⌈

Kind:	class
Header file:	#include "ara/log/log_stream.h"
Forwarding header file:	#include "ara/log/log_fwd.h"
Scope:	namespace ara::log
Symbol:	LogStream
Syntax:	`class LogStream final {...};`
Description:	Class LogStream Initiates it with the given log level directly on the back-end. .

⌋(*RS_LT_00003*)

Note:
Normally `Application process`es would not use this class directly. Instead one of the log methods provided in the main `Logging API` shall be used. Those methods automatically setup a temporary object of this class with the given log severity level. The only reason to use this class directly is, if the user wants to hold a `ara::log::LogStream` object longer than the default one-statement scope. This is useful in order to create log messages that are distributed over multiple code lines. See the `ara::log::LogStream::Flush` method for further information. Once this temporary object gets out of scope, its destructor takes care that the message buffer is ready to be processed by the Logging framework.

8.3.1.1 Extending the Logging API to understand custom types

The `ara::log::LogStream` class supports natively the formats stated in chapter 8.2, it can be easily extended for other derived types by providing a stream operator that makes use of already supported types.

Example:

```
struct MyCustomType {
  int8_t foo;
  ara::core::String bar;
};

LogStream& operator<<(LogStream& out, const MyCustomType& value) {
  return (out << value.foo << value.bar);
}

// Producing the output "42 the answer is."
Logger& ctx0 = CreateLogger("CTX0", "Context Description CTX0");
```

```
12 ctx0.LogDebug () << MyCustomType{42, " the answer is."};
```

8.3.1.2 LogStream::Flush

[SWS_LOG_00039]{DRAFT} **Definition of API function ara::log::LogStream::Flush** ⌈

Kind:	function
Header file:	#include "ara/log/log_stream.h"
Scope:	class ara::log::LogStream
Symbol:	Flush()
Syntax:	`void Flush () noexcept;`
Return value:	None
Exception Safety:	noexcept
Thread Safety:	reentrant
Description:	Sends out the current log buffer and initiates a new message stream. .

⌋(*RS_LT_00003*)

Note:
Calling `ara::log::LogStream::Flush` is only necessary if the `ara::log::LogStream` object is going to be re-used within the same scope. Otherwise, if the object goes out of scope (e.g. end of function block) then the flushing operation will be done internally by the destructor. It is important to note that the `ara::log::LogStream::Flush` command does not empty the buffer, but it forwards the buffer's current contents to the `Logging framework`.

8.3.1.3 Built-in operators for natively supported types

[SWS_LOG_00040]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(bool value)	
Syntax:	`LogStream & operator<< (bool value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Appends given value to the internal message buffer.	

⌋(*RS_LT_00003*)

[SWS_LOG_00041]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::uint8_t value)	
Syntax:	`LogStream & operator<< (std::uint8_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes unsigned int 8 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00042]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::uint16_t value)	
Syntax:	`LogStream & operator<< (std::uint16_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes unsigned int 16 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00043]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::uint32_t value)	
Syntax:	`LogStream & operator<< (std::uint32_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes unsigned int 32 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00044]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::uint64_t value)	
Syntax:	`LogStream & operator<< (std::uint64_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes unsigned int 64 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00045]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::int8_t value)	
Syntax:	`LogStream & operator<< (std::int8_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes signed int 8 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00046]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::int16_t value)	
Syntax:	`LogStream & operator<< (std::int16_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes signed int 16 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00047]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::int32_t value)	
Syntax:	`LogStream & operator<< (std::int32_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes signed int 32 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00048]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(std::int64_t value)	
Syntax:	`LogStream & operator<< (std::int64_t value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes signed int 64 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00049]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(float value)	
Syntax:	`LogStream & operator<< (float value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes float 32 bit parameter into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00050]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(double value)	
Syntax:	`LogStream & operator<< (double value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes float 64 bit parameter into message.	

⌋(*RS_LT_00003*)

8.3.1.4 Built-in operators for extra types

[SWS_LOG_00062]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(const ara::core::StringView value)	
Syntax:	`LogStream & operator<< (const ara::core::StringView value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Writes ara::core::StringView into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00051]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(const char *const value)	
Syntax:	`LogStream & operator<< (const char *const value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	

▽

△

Thread Safety:	reentrant
Description:	Writes null terminated UTF8 string into message.

⌋(*RS_LT_00003*)

[SWS_LOG_00063]{DRAFT} Definition of API function ara::log::operator<< ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	namespace ara::log	
Symbol:	operator<<(LogStream &out, LogLevel value)	
Syntax:	`LogStream & operator<< (LogStream &out, LogLevel value) noexcept;`	
Parameters (in):	out	LogStream Object which is used to append the logged LogLevel (value) to
	value	LogLevel enum parameter as text to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Description:	Appends LogLevel enum parameter as text into message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00124]{DRAFT} Definition of API function ara::log::operator<< ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	namespace ara::log	
Symbol:	operator<<(LogStream &out, const core::ErrorCode &ec)	
Syntax:	`LogStream & operator<< (LogStream &out, const core::ErrorCode &ec) noexcept;`	
Parameters (in):	out	the LogStream object into which to add the value
	ec	the ErrorCode instance to log
Return value:	LogStream &	out
Exception Safety:	noexcept	
Description:	Write a core::ErrorCode instance into the message. When output to the console, the ErrorCode shall be shown in an implementation-defined way as a String holding the result of ErrorCode:Domain().Name() (i.e. the ErrorDomain's Shortname), and the integral error code number.	

⌋(*RS_LT_00003*)

[SWS_LOG_00125]{DRAFT} Definition of API function ara::log::operator<< ⌈

Kind:	function
Header file:	#include "ara/log/log_stream.h"
Scope:	namespace ara::log
Symbol:	operator<<(LogStream &out, const std::chrono::duration< Rep, Period > &value)

▽

△

Syntax:	`template <typename Rep, typename Period>` `LogStream & operator<< (LogStream &out, const std::chrono::duration< Rep, Period > &value) noexcept;`	
Template param:	Rep	arithmetic type representing the number of ticks in this duration
	Period	a std::ratio type representing the tick period of the clock, in seconds
Parameters (in):	out	the LogStream object into which to add the value
	value	the duration instance to log
Return value:	LogStream &	out
Exception Safety:	noexcept	
Description:	Write a std::chrono::duration instance into the message.	

⌋(*RS_LT_00049*)

[SWS_LOG_00126]{DRAFT} **Definition of API function ara::log::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	namespace ara::log	
Symbol:	operator<<(LogStream &out, const ara::core::InstanceSpecifier &value)	
Syntax:	`LogStream & operator<< (LogStream &out, const ara::core::Instance Specifier &value) noexcept;`	
Parameters (in):	out	the LogStream object into which to add the value
	value	the InstanceSpecifier to log
Return value:	LogStream &	out
Exception Safety:	noexcept	
Description:	Write a core::InstanceSpecifier into the message.	

⌋(*RS_LT_00049*)

[SWS_LOG_00127]{DRAFT} **Definition of API function ara::log::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	namespace ara::log	
Symbol:	operator<<(LogStream &out, const void *value)	
Syntax:	`LogStream & operator<< (LogStream &out, const void *value) noexcept;`	
Parameters (in):	out	the LogStream object into which to add the value
	value	to log
Return value:	LogStream &	out
Exception Safety:	noexcept	

⌋(*RS_LT_00049*)

[SWS_LOG_00128]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(core::Span< const core::Byte > data)	
Syntax:	`LogStream & operator<< (core::Span< const core::Byte > data) noexcept;`	
Parameters (in):	data	a Span<const Byte> covering the range to be logged
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Description:	Write a byte sequence into message. This call shall copy the sequence of core::Byte objects as-is into the message.	

⌋(*RS_LT_00003*)

8.3.1.5 Attribute handling

[SWS_LOG_00203]{DRAFT} **Definition of API function ara::log::LogStream::operator<<** ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	operator<<(const Argument< T > &arg)	
Syntax:	`template <typename T>` `LogStream & operator<< (const Argument< T > &arg) noexcept;`	
Template param:	T	the argument payload type
Parameters (in):	arg	the argument wrapper object
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Description:	Log an argument with attributes. When output to the console, the value and all its attributes shall be shown as a single argument.	

⌋(*RS_LT_00003*)

8.3.1.6 Modifiers

[SWS_LOG_00258]{DRAFT} **Definition of API function ara::log::LogStream::WithPrivacy** ⌈

Kind:	function
Header file:	#include "ara/log/log_stream.h"
Scope:	class ara::log::LogStream

▽

△

Symbol:	WithPrivacy(T value)	
Syntax:	`template <typename T>` `LogStream & WithPrivacy (T value) noexcept;`	
Parameters (in):	value	a (project-specific) value to add as privacy level of the message template
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Description:	Set the message's privacy level. A program that calls this function with a T that is neither an integral nor an enum type is ill-formed. Only the lower 8 bits of value are used, any higher-level bits are ignored.	

⌋(*RS_LT_00003*)

[SWS_LOG_00129]{DRAFT} Definition of API function ara::log::LogStream::WithLocation ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	WithLocation(core::StringView file, int line)	
Syntax:	`LogStream & WithLocation (core::StringView file, int line) noexcept;`	
Parameters (in):	file	the source file identifier
	line	the source file line number
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Description:	Add source file location into the message.	

⌋(*RS_LT_00003*)

[SWS_LOG_00132]{DRAFT} Definition of API function ara::log::LogStream::WithTag ⌈

Kind:	function	
Header file:	#include "ara/log/log_stream.h"	
Scope:	class ara::log::LogStream	
Symbol:	WithTag(core::StringView tag)	
Syntax:	`LogStream & WithTag (core::StringView tag) noexcept;`	
Parameters (in):	tag	the tag text to attach to the current message
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Description:	Add the given single tag to the current message.	

⌋(*RS_LT_00003*)

8.3.2 Class Logger

The class `ara::log::Logger` represents a logger context. The `Logging framework` defines contexts which can be seen as logger instances within one `Application process` or process scope.

[SWS_LOG_00172]{DRAFT} **Definition of API class ara::log::Logger** ⌈

Kind:	class
Header file:	#include "ara/log/logger.h"
Forwarding header file:	#include "ara/log/log_fwd.h"
Scope:	namespace ara::log
Symbol:	Logger
Syntax:	`class Logger final {...};`
Description:	Interface for sending log messages.

⌋*(RS_LT_00003)*

The contexts have the following properties:

1) Context ID
2) Description of the Context ID
3) Default log level

A context will be automatically registered against the `Logging back-end` during creation phase, as well as automatically deregistered during process shutdown phase. So the end user does not care for the objects life time. To ensure such housekeeping functionality, a strong ownership of the logger instances needs to be ensured towards the `Logging framework`. This means that the `Application process` are not supposed to call the Logger constructor themselves.

The user is not allowed to create a Logger object by himself. Logger context needs to be created by the provided API call `CreateLogger()`.

8.3.2.1 Logger::LogFatal

[SWS_LOG_00064]{DRAFT} **Definition of API function ara::log::Logger::LogFatal** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogFatal()	
Syntax:	`LogStream LogFatal () const noexcept;`	
Return value:	LogStream	LogStream object of Fatal severity.

▽

△

Exception Safety:	noexcept
Description:	Creates a `LogStream` object. Returned object will accept arguments via the insert stream operator "@c <<".
Notes:	In the normal usage scenario, the object's life time of the created LogStream is scoped within one statement (ends with ; after last passed argument). If one wants to extend the LogStream object's life time, the object might be assigned to a named variable.

⌋(*RS_LT_00003*)

8.3.2.2 Logger::LogError

[SWS_LOG_00065]{DRAFT} **Definition of API function ara::log::Logger::LogError**
⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogError()	
Syntax:	`LogStream LogError () const noexcept;`	
Return value:	LogStream	LogStream object of Error severity.
Exception Safety:	noexcept	
Description:	Same as `Logger::LogFatal()`.	

⌋(*RS_LT_00003*)

8.3.2.3 Logger::LogWarn

[SWS_LOG_00066]{DRAFT} **Definition of API function ara::log::Logger::LogWarn**
⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogWarn()	
Syntax:	`LogStream LogWarn () const noexcept;`	
Return value:	LogStream	LogStream object of Warn severity.
Exception Safety:	noexcept	
Description:	Same as `Logger::LogFatal()`.	

⌋(*RS_LT_00003*)

8.3.2.4 Logger::LogInfo

[SWS_LOG_00067]{DRAFT} **Definition of API function ara::log::Logger::LogInfo** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogInfo()	
Syntax:	`LogStream LogInfo () const noexcept;`	
Return value:	LogStream	LogStream object of Info severity.
Exception Safety:	noexcept	
Description:	Same as `Logger::LogFatal()`.	

⌋(*RS_LT_00003*)

8.3.2.5 Logger::LogDebug

[SWS_LOG_00068]{DRAFT} **Definition of API function ara::log::Logger::LogDebug** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogDebug()	
Syntax:	`LogStream LogDebug () const noexcept;`	
Return value:	LogStream	LogStream object of Debug severity.
Exception Safety:	noexcept	
Description:	Same as `Logger::LogFatal()`.	

⌋(*RS_LT_00003*)

8.3.2.6 Logger::LogVerbose

[SWS_LOG_00069]{DRAFT} **Definition of API function ara::log::Logger::LogVerbose** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogVerbose()	
Syntax:	`LogStream LogVerbose () const noexcept;`	
Return value:	LogStream	LogStream object of Verbose severity.
Exception Safety:	noexcept	
Description:	Same as `Logger::LogFatal()`.	

⌋(*RS_LT_00003*)

8.3.2.7 Logger::IsEnabled

[SWS_LOG_00070]{DRAFT} **Definition of API function ara::log::Logger::IsEnabled** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	IsEnabled(LogLevel logLevel)	
Syntax:	`bool IsEnabled (LogLevel logLevel) const noexcept;`	
Parameters (in):	logLevel	The to be checked log level.
Return value:	bool	True if desired log level satisfies the configured reporting level.
Exception Safety:	noexcept	
Description:	Check current configured log reporting level. Applications may want to check the actual configured reporting log level of certain loggers before doing log data preparation that is runtime intensive.	

⌋(*RS_LT_00003*)

8.3.2.8 Logger::WithLevel

[SWS_LOG_00131]{DRAFT} **Definition of API function ara::log::Logger::WithLevel** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	WithLevel(LogLevel logLevel)	
Syntax:	`LogStream WithLevel (LogLevel logLevel) const noexcept;`	
Parameters (in):	logLevel	the log level to use for this LogStream instance
Return value:	LogStream	a new LogStream instance with the given log level
Exception Safety:	noexcept	
Description:	Log message with a programmatically determined log level can be written.	

⌋(*RS_LT_00003*)

8.3.2.9 Logger::Log

[SWS_LOG_00204]{DRAFT} **Definition of API function ara::log::Logger::Log** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	Log(const MsgId &id, const Params &... args)	
Syntax:	`template <typename MsgId, typename... Params>` `void Log (const MsgId &id, const Params &... args) noexcept;`	
Template param:	MsgId	the type of the id parameter
	Args	the types of the args parameters
Parameters (in):	id	an implementation-defined type identifying the message object
	args	the arguments to add to the message
Return value:	None	
Exception Safety:	noexcept	
Description:	Log a modeled message. If this function is called with an argument list that does not match the modeled message, the program is ill-formed.	

⌋(*RS_LT_00003*)

8.3.2.10 Logger::LogWith

[SWS_LOG_00133]{DRAFT} **Definition of API function ara::log::Logger::LogWith** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	LogWith(const std::tuple< Attrs... > &attrs, const MsgId &msgId, const Params &... params)	
Syntax:	`template <typename... Attrs, typename MsgId, typename... Params>` `void LogWith (const std::tuple< Attrs... > &attrs, const MsgId &msgId, const Params &... params) noexcept;`	
Template param:	Attrs	the types of attrs
	MsgId	the type of msgId
	Params	the types of the params
Parameters (in):	attrs	a std::tuple containing attributes of the message
	msgId	an implementation-defined instance of type MsgId identifying the message object
	params	the arguments to add to the message
Return value:	None	
Exception Safety:	noexcept	
Description:	This function does not participate in overload resolution if any of Attrs... is not an "Attr" (TBD).	

⌋(*RS_LT_00003*)

8.3.2.11 Logger::SetThreshold

[SWS_LOG_00255]{DRAFT} **Definition of API function ara::log::Logger::SetThreshold** ⌈

Kind:	function	
Header file:	#include "ara/log/logger.h"	
Scope:	class ara::log::Logger	
Symbol:	SetThreshold(LogLevel threshold)	
Syntax:	`void SetThreshold (LogLevel threshold) noexcept;`	
Parameters (in):	threshold	the new threshold
Return value:	None	
Exception Safety:	noexcept	
Description:	Set log level threshold for this Logger instance.	

⌋(*RS_LT_00049*)

8.3.2.12 Wrapper type

[SWS_LOG_00261] Definition of API class ara::log::Argument ⌈

Kind:	class	
Header file:	#include "ara/log/log_stream.h"	
Forwarding header file:	#include "ara/log/log_fwd.h"	
Scope:	namespace ara::log	
Symbol:	Argument	
Syntax:	`template <typename T>` `class Argument final {...};`	
Template param:	typename T	the argument payload type
Description:	Wrapper type for holding a payload argument with its attributes. The setup of this class is implementation-defined.	

⌋(*RS_LT_00003*)

A Mentioned Manifest Elements

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document.

Class	**DltApplication**			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	This meta-class represents the application from which the log and trace message originates.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	DltEcu.application			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
applicationDescription	String	0..1	attr	This attribute can be used to describe the applicationId that is used in the log and trace message in more detail.
applicationId	String	0..1	attr	This attribute identifies the SW-C/BSW module in the log and trace message.
context	DltContext	*	ref	Definition of ContextIds for the Application. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=context.dltContext, context.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime

Table A.1: DltApplication

Class	**DltArgument**			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	This element defines an Argument in a DltMessage.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	DltArgument.dltArgumentEntry, DltMessage.dltArgument			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dltArgumentEntry	DltArgument	*	aggr	This aggregation is used to describe subElements of a Dlt Argument that defines a Structure.
length	PositiveInteger	0..1	attr	Describes the DltArgument length in case of Arrays and Strings in number of BaseTypes.
networkRepresentation	SwDataDefProps	0..1	aggr	Definition of the networkRepresentation of the Dlt Argument. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=networkRepresentation
optional	Boolean	0..1	attr	This attribute defines whether the argument is optional or not. If set to true, the argument can be omitted from the payload of a DLT message.
predefinedText	Boolean	0..1	attr	This attribute defines whether the DltArgument is a predefinedText (Static Data).
variableLength	Boolean	0..1	attr	This attribute defines whether the length of the Dlt Argument is variable (determined at runtime) or not.

Table A.2: DltArgument

Class	**DltContext**			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	This meta-class represents the Context that groups Log and Trace Messages that are generated by an application. **Tags:** atp.recommendedPackage=DltContexts			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
context Description	String	0..1	attr	This attribute can be used to describe the contextId that is used in the log and trace message in more detail.
contextId	String	0..1	attr	This attribute is used to group log and trace messages produced by an application to distinguish functionality.
dltMessage	DltMessage	*	ref	Group of Log and Trace Messages assigned to the Dlt Context **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=dltMessage.dltMessage, dlt Message.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime

Table A.3: DltContext

Class	**DltEcu**			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	This element represents an Ecu or Machine that produces logging and tracing information. **Tags:** atp.recommendedPackage=DltEcus			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
application	DltApplication	*	aggr	Application on DltEcu that provides log or trace data. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=application.shortName, application.variation Point.shortLabel vh.latestBindingTime=systemDesignTime
ecuId	String	0..1	attr	This attribute defines the name of the ECU for use within the Dlt protocol.

Table A.4: DltEcu

Class	**DltLogSink**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::LogAndTrace			
Note	The meta-class defines the output sink for DltLogMessages **Tags:** atp.recommendedPackage=DltLogSinks			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
bufferOutput	Boolean	0..1	attr	This attribute defines whether a buffer is used in case that the output sink is the console.

▽

△

Class	**DltLogSink**			
defaultLog Threshold	LogTraceDefaultLog LevelEnum	0..1	attr	This attribute allows to set a log level Threshold for Log Level filtering.
defaultTrace State	Boolean	0..1	attr	This attributes defines the default trace status.
endpoint Configuration	PlatformModule EthernetEndpoint Configuration	0..1	ref	Network configuration (Protocol, Port, IP Address) for transmission of dlt messages on a specific VLAN.
logChannelId	String	0..1	attr	This attribute identifies the LogChannel for usage within the Log And Trace protocol.
nonVerbose Mode	Boolean	0..1	attr	This attribute defines whether this DltLogSink supports non-Verbose Dlt messages. If disabled only verbose mode messages shall be used.
path	UriString	0..1	attr	This attribute defines the path to the file that is used as output sink.
queueSize	PositiveInteger	0..1	attr	Length of the queue (in which messages can be stored before processing) in the unit "Log message".
segmentation Supported	Boolean	0..1	attr	If enabled, segmentation will be used for DLT messages that are larger than EthernetCommunication Connector.maximumTransmissionUnit referenced via Dlt LogSink.endpointConfiguration.

Table A.5: DltLogSink

Class	**DltLogSinkToPortPrototypeMapping**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::LogAndTrace			
Note	This meta-class maps a PortPrototype to an output sink of a log and trace message. **Tags:** atp.recommendedPackage=DltLogSinkToPortPrototypeMappings			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dltContext	DltContext	0..1	ref	Assignement of the DltContext that groups log and trace messages that will be transmitted to the DltLogSink.
dltLogSink	DltLogSink	*	ref	Reference to the output sink to which the log or trace message will be transmitted,
dltSessionId	PositiveInteger	0..1	attr	This attribute allows distinguishing log/trace messages from different instances of the same SW-C.
pPortPrototype	PPortPrototype	0..1	iref	Reference to PPortPrototype that is mapped to the DltLog Sink. **InstanceRef implemented by:** PPortPrototypeIn ExecutableInstanceRef
process	Process	0..1	ref	This reference represents the process required as context for the mapping.
rPortPrototype	RPortPrototype	0..1	iref	Reference to RPortPrototype that is mapped to a DltLog Sink **InstanceRef implemented by:** RPortPrototypeIn ExecutableInstanceRef

Table A.6: DltLogSinkToPortPrototypeMapping

Class	**DltMessage**			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	This element defines a DltMessage.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	LogAndTraceMessageCollectionSet.dltMessage			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dltArgument (ordered)	DltArgument	*	aggr	Ordered collection of DltArguments in the DltMessage.
messageId	PositiveInteger	0..1	attr	This attribute defines the unique Id for the DltMessage.
messageLine Number	PositiveInteger	0..1	attr	This attribute describes the position in the source file in which this log message was called.
messageSource File	String	0..1	attr	This attribute describes the source file in which this log message was called.
messageType Info	String	0..1	attr	This attribute describes the message Type
privacyLevel	PrivacyLevel	0..1	aggr	The Privacy Level helps to identify the Log and Trace content towards the degree of privacy to it.

Table A.7: DltMessage

Class	**Executable**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::ApplicationStructure			
Note	This meta-class represents an executable program. **Tags:** atp.recommendedPackage=Executables			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
buildType	BuildTypeEnum	0..1	attr	This attribute describes the buildType of a module and/or platform implementation.
implementation Props	Executable ImplementationProps	*	aggr	This aggregation contains the collection of implementation-specific properties necessary to properly build the enclosing Executable.
minimumTimer Granularity	TimeValue	0..1	attr	This attribute describes the minimum timer resolution (TimeValue of one tick) that is required by the Executable.
reporting Behavior	ExecutionState ReportingBehavior Enum	0..1	attr	this attribute controls the execution state reporting behavior of the enclosing Executable.
rootSw Component Prototype	RootSwComponent Prototype	0..1	aggr	This represents the root SwCompositionPrototype of the Executable. This aggregation is required (in contrast to a direct reference of a SwComponentType) in order to support the definition of instanceRefs in Executable context.
traceSwitch Configuration	TraceSwitch Configuration	*	aggr	Configuration of the MsgId based trace switch **Tags:** atp.Status=draft
version	StrongRevisionLabel String	0..1	attr	Version of the executable.

Table A.8: Executable

Class	***FunctionalClusterInteractsWithFunctionalClusterMapping*** (abstract)			
Package	M2::AUTOSARTemplates::AdaptivePlatform::FunctionalClusterInteractsWithFunctionalClusterMapping			
Note	This meta-class identifies a relation between functional clusters on the adaptive platform such one functional cluster can call APIs of the other functional cluster.			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Subclasses	ArtifactChecksumToCryptoProviderMapping, ComCertificateToCryptoCertificateMapping, ComKeyTo CryptoKeySlotMapping, ComSecOcToCryptoKeySlotMapping, FunctionalClusterInteractsWithPersistency DeploymentMapping, FunctionalClusterToSecurityEventDefinitionMapping, NmInteractsWithSmMapping, PersistencyDeploymentElementToCryptoKeySlotMapping, PersistencyDeploymentToCryptoKeySlot Mapping, PersistencyDeploymentToDltLogSinkMapping, SmInteractsWithNmMapping, TimeBase ProviderToPersistencyMapping, UcmToTimeBaseResourceMapping			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
–	–	–	–	–

Table A.9: FunctionalClusterInteractsWithFunctionalClusterMapping

Class	***Identifiable*** (abstract)
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable
Note	Instances of this class can be referred to by their identifier (within the namespace borders). In addition to this, Identifiables are objects which contribute significantly to the overall structure of an AUTOSAR description. In particular, Identifiables might contain Identifiables.
Base	*ARObject*, *MultilanguageReferrable*, *Referrable*
Subclasses	ARPackage, *AbstractDoIpLogicAddressProps*, *AbstractEvent*, *AbstractImplementationDataTypeElement*, *AbstractSecurityEventFilter*, *AbstractSecurityIdsmInstanceFilter*, *AbstractServiceInstance*, *Abstract SignalBasedToISignalTriggeringMapping*, AdaptiveSwcInternalBehavior, ApApplicationEndpoint, ApplicationEndpoint, ApplicationError, AppliedStandard, ArtifactChecksum, ArtifactLocator, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpFeature*, AutosarOperationArgumentInstance, AutosarVariable Instance, *BuildActionEntity*, BuildActionEnvironment, Chapter, CheckpointTransition, ClassContent Conditional, ClientIdDefinition, ClientServerOperation, Code, *CollectableElement*, ComManagement Mapping, *CommConnectorPort*, *CommunicationConnector*, *CommunicationController*, Compiler, ConsistencyNeeds, ConsumedEventGroup, CouplingPort, *CouplingPortStructuralElement*, Crypto Certificate, CryptoKeySlot, CryptoProvider, *CryptoServiceMapping*, DataPrototypeGroup, Data Transformation, DdsCpDomain, DdsCpPartition, DdsCpQosProfile, DdsCpTopic, DdsDomainRange, DependencyOnArtifact, *DiagEventDebounceAlgorithm*, DiagnosticAuthTransmitCertificateEvaluation, DiagnosticConnectedIndicator, DiagnosticDataElement, DiagnosticDebounceAlgorithmProps, Diagnostic FunctionInhibitSource, DiagnosticParameterElement, *DiagnosticRoutineSubfunction*, DiagnosticSovd MethodPrimitive, DltApplication, DltArgument, DltMessage, DoIpInterface, DoIpLogicAddress, DoIp RoutingActivation, E2EProfileConfiguration, End2EndEventProtectionProps, End2EndMethodProtection Props, EndToEndProtection, EthernetWakeupSleepOnDatalineConfig, EventHandler, EventMapping, ExclusiveArea, *ExecutableEntity*, *ExecutionTime*, FMAttributeDef, FMFeatureMapAssertion, FMFeature MapCondition, FMFeatureMapElement, FMFeatureRelation, FMFeatureRestriction, FMFeatureSelection, FieldMapping, FireAndForgetMethodMapping, FlexrayArTpNode, FlexrayTpPduPool, *FrameTriggering*, GeneralParameter, GlobalSupervision, GlobalTimeGateway, *GlobalTimeMaster*, *GlobalTimeSlave*, *HealthChannel*, *HeapUsage*, HwAttributeDef, HwAttributeLiteralDef, HwPin, HwPinGroup, *IEEE1722Tp AcfBus*, *IEEE1722TpAcfBusPart*, IPSecRule, IPv6ExtHeaderFilterList, ISignalToIPduMapping, ISignal Triggering, *IdentCaption*, ImpositionTime, InternalTriggeringPoint, Keyword, LifeCycleState, Linker, Mac MulticastGroup, MacSecKayParticipant, McDataInstance, MemorySection, MemoryUsage, Method Mapping, ModeDeclaration, ModeDeclarationMapping, ModeSwitchPoint, NetworkEndpoint, *NmCluster*, *NmNode*, *PackageableElement*, ParameterAccess, PduActivationRoutingGroup, PduToFrameMapping, PduTriggering, PerInstanceMemory, *PersistencyDeploymentElement*, *PersistencyInterfaceElement*, *Phm Supervision*, *PhysicalChannel*, PortGroup, *PortInterfaceMapping*, PossibleErrorReaction, ProcessTo MachineMapping, Processor, ProcessorCore, PskIdentityToKeySlotMapping, ResourceConsumption, ResourceGroup, RootSwClusterDesignComponentPrototype, RootSwComponentPrototype, RootSw CompositionPrototype, RptComponent, RptContainer, RptExecutableEntity, RptExecutableEntityEvent, RptExecutionContext, RptProfile, RptServicePoint, RunnableEntityGroup, *SdgAttribute*, SdgClass, Sec OcJobMapping, SecOcJobRequirement, SecureCommunicationAuthenticationProps, *Secure CommunicationDeployment*, SecureCommunicationFreshnessProps, SecurityEventContextProps, ▽

▽

△

Class	***Identifiable*** (abstract)			
	△ *ServiceEventDeployment*, *ServiceFieldDeployment*, ServiceInterfaceElementSecureComConfig, *ServiceMethodDeployment*, *ServiceNeeds*, SignalServiceTranslationEventProps, SignalServiceTranslationProps, SocketAddress, SoftwarePackageStep, SomeipEventGroup, SomeipProvidedEventGroup, SomeipTpChannel, *SpecElementReference*, *StackUsage*, *StateManagementActionItem*, StateManagementActionList, StateManagementStateNotification, *StateManagementStateRequest*, StaticSocketConnection, StructuredReq, SupervisionCheckpoint, SupervisionMode, SupervisionModeCondition, SwGenericAxisParamType, SwServiceArg, SwcServiceDependency, SystemMapping, *TimeBaseResource*, *TimingClock*, TimingClockSyncAccuracy, TimingCondition, *TimingConstraint*, *TimingDescription*, TimingExtensionResource, TimingModeInstance, TlsCryptoCipherSuite, TlsCryptoCipherSuiteProps, TlsJobMapping, Topic1, TpAddress, TraceableTable, TraceableText, *TracedFailure*, *TransformationProps*, TransformationTechnology, Trigger, UcmDescription, UcmRetryStrategy, UcmStep, VariableAccess, VariationPointProxy, VehicleRolloutStep, ViewMap, VlanConfig, WaitPoint			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
adminData	AdminData	0..1	aggr	This represents the administrative data for the identifiable object. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=adminData xml.sequenceOffset=-40
annotation	Annotation	*	aggr	Possibility to provide additional notes while defining a model element (e.g. the ECU Configuration Parameter Values). These are not intended as documentation but are mere design notes. **Tags:** xml.sequenceOffset=-25
category	CategoryString	0..1	attr	The category is a keyword that specializes the semantics of the Identifiable. It affects the expected existence of attributes and the applicability of constraints. **Tags:** xml.sequenceOffset=-50
desc	MultiLanguageOverviewParagraph	0..1	aggr	This represents a general but brief (one paragraph) description what the object in question is about. It is only one paragraph! Desc is intended to be collected into overview tables. This property helps a human reader to identify the object in question. More elaborate documentation, (in particular how the object is built or used) should go to "introduction". **Tags:** xml.sequenceOffset=-60
introduction	DocumentationBlock	0..1	aggr	This represents more information about how the object in question is built or is used. Therefore it is a DocumentationBlock. **Tags:** xml.sequenceOffset=-30
uuid	String	0..1	attr	The purpose of this attribute is to provide a globally unique identifier for an instance of a meta-class. The values of this attribute should be globally unique strings prefixed by the type of identifier. For example, to include a DCE UUID as defined by The Open Group, the UUID would be preceded by "DCE:". The values of this attribute may be used to support merging of different AUTOSAR models. The form of the UUID (Universally Unique Identifier) is taken from a standard defined by the Open Group (was Open Software Foundation). This standard is widely used, including by Microsoft for COM (GUIDs) and by many companies for DCE, which is based on CORBA. The method for generating these 128-bit IDs is published in the standard and the effectiveness and uniqueness of the IDs is not in practice disputed. If the id namespace is omitted, DCE is assumed. An example is "DCE:2fac1234-31f8-11b4-a222-08002b34c003". The ▽

▽

△

Class	Identifiable (abstract)			
				△ uuid attribute has no semantic meaning for an AUTOSAR model and there is no requirement for AUTOSAR tools to manage the timestamp. **Tags:** xml.attribute=true

Table A.10: Identifiable

Class	LogAndTraceInstantiation			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::LogAndTrace			
Note	This meta-class defines the attributes for the Log&Trace configuration on a specific machine.			
Base	*ARObject*, *AdaptiveModuleInstantiation*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *NonOsModuleInstantiation*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, Machine.moduleInstantiation			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
dltEcu	DltEcu	0..1	ref	Reference to the Ecu representation in the Log And Trace Extract.
logSink	DltLogSink	*	ref	Reference to output sinks for log or trace messages that are produced on the Machine.
sessionId Support	Boolean	0..1	attr	This attribute defines whether the sessionId is used or not.
timeBase Resource	TimeBaseResource	*	ref	This reference is used to describe to which time base the Log and Trace module has access. From the Time Base Resource the Log and Trace module gets the needed information to generate the time stamp.

Table A.11: LogAndTraceInstantiation

Class	LogAndTraceInterface			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			
Note	This meta-class provides the ability to implement a PortInterface for support of Logging or Tracing. **Tags:** atp.recommendedPackage=PortInterfaces			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PortInterface*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
–	–	–	–	–

Table A.12: LogAndTraceInterface

Class	LogAndTraceMessageCollectionSet			
Package	M2::AUTOSARTemplates::LogAndTraceExtract			
Note	Collection of DltMessages **Tags:** atp.recommendedPackage=LogAndTraceMessageCollectionSets			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**

▽

△

Class	**LogAndTraceMessageCollectionSet**			
dltMessage	DltMessage	*	aggr	Collection of DltMessages in the DltMessageCollection Set. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=dltMessage.shortName, dlt Message.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime

Table A.13: LogAndTraceMessageCollectionSet

Class	**Machine**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::MachineManifest			
Note	Machine that represents an Adaptive Autosar Software Stack. **Tags:** atp.recommendedPackage=Machines			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeployment Element*, *UploadablePackageElement*			
Aggregated by	ARPackage.element, *AtpClassifier*.atpFeature			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
default Application Timeout	EnterExitTimeout	0..1	aggr	This aggregation defines a default timeout in the context of a given Machine with respect to the launching and termination of applications.
environment Variable	TagWithOptionalValue	*	aggr	This aggregation represents the collection of environment variables that shall be added to the environment defined on the level of the enclosing Machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=environmentVariable
machineDesign	MachineDesign	0..1	ref	Reference to the MachineDesign this Machine is implementing.
module Instantiation	AdaptiveModule Instantiation	*	aggr	Configuration of Adaptive Autosar module instances that are running on the machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=moduleInstantiation.shortName
processor	Processor	*	aggr	This represents the collection of processors owned by the enclosing machine.
secure Communication Deployment	SecureCommunication Deployment	*	aggr	Deployment of secure communication protocol configuration settings to crypto module entities. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=secureCommunication Deployment.shortName
trustedPlatform Executable LaunchBehavior	TrustedPlatform ExecutableLaunch BehaviorEnum	0..1	attr	This attribute controls the behavior of how authentication affects the ability to launch for each Executable.

Table A.14: Machine

Class	**PlatformModuleEthernetEndpointConfiguration**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::AdaptiveModule Implementation			
Note	This meta-class defines the attributes for the configuration of a port, protocol type and IP address of the communication on a VLAN. **Tags:** atp.recommendedPackage=PlatformModuleEndpointConfigurations			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *PlatformModuleEndpointConfiguration*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
communication Connector	EthernetCommunication Connector	0..1	ref	Reference to the CommunicationConnector (VLAN) for which the network configuration is defined.
ipv4MulticastIp Address	Ip4AddressString	0..1	attr	Multicast IPv4 Address to which the message will be transmitted.
ipv6MulticastIp Address	Ip6AddressString	0..1	attr	Multicast IPv6 Address to which the message will be transmitted.
secureCom PropsForTcp	SecureComProps	0..1	ref	Reference to communication security configuration settings that are valid for the tcp unicast endpoint (Tcp Port + unicast IP Address) defined by the PlatformModule EthernetEndpointConfiguration.
secureCom PropsForUdp	SecureComProps	0..1	ref	Reference to communication security configuration settings that are valid for the udp unicast endpoint (Udp Port + unicast IP Address) defined by the PlatformModule EthernetEndpointConfiguration.
tcpPort	ApApplicationEndpoint	0..1	ref	This reference allows to configure a tcp port number.
udpPort	ApApplicationEndpoint	0..1	ref	This reference allows to configure a udp port number.

Table A.15: PlatformModuleEthernetEndpointConfiguration

Class	***PortPrototype*** (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Base class for the ports of an AUTOSAR software component. The aggregation of PortPrototypes is subject to variability with the purpose to support the conditional existence of ports.			
Base	*ARObject*, *AtpBlueprintable*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	*AbstractProvidedPortPrototype*, *AbstractRequiredPortPrototype*			
Aggregated by	*AtpClassifier*.atpFeature, *SwComponentType*.port			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
clientServer Annotation	ClientServerAnnotation	*	aggr	Annotation of this PortPrototype with respect to client/ server communication.
delegatedPort Annotation	DelegatedPort Annotation	0..1	aggr	Annotations on this delegated port.
ioHwAbstraction Server Annotation	IoHwAbstractionServer Annotation	*	aggr	Annotations on this IO Hardware Abstraction port.
modePort Annotation	ModePortAnnotation	*	aggr	Annotations on this mode port.
nvDataPort Annotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port.
parameterPort Annotation	ParameterPort Annotation	*	aggr	Annotations on this parameter port.
portPrototype Props	PortPrototypeProps	0..1	aggr	This attribute allows for the definition of further qualification of the semantics of a PortPrototype.

△

Class	***PortPrototype*** (abstract)			
senderReceiver Annotation	SenderReceiver Annotation	*	aggr	Collection of annotations of this ports sender/receiver communication.
triggerPort Annotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port.

Table A.16: PortPrototype

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			
Note	This meta-class provides information required to execute the referenced `Executable`. **Tags:** atp.recommendedPackage=Processes			
Base	*ARElement*, *ARObject*, *AbstractExecutionContext*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeploymentElement*, *Uploadable PackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
design	ProcessDesign	0..1	ref	This reference represents the identification of the design-time representation for the Process that owns the reference.
executable	Executable	*	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef
functionCluster Affiliation	String	0..1	attr	This attribute specifies which functional cluster the Process is affiliated with.
numberOf RestartAttempts	PositiveInteger	0..1	attr	This attribute defines how often a process shall be restarted if the start fails. numberOfRestartAttempts = "0" OR Attribute not existing, start once numberOfRestartAttempts = "1", start a second time
preMapping	Boolean	0..1	attr	This attribute describes whether the executable is preloaded into the memory.
processState Machine	ModeDeclarationGroup Prototype	0..1	aggr	Set of Process States that are defined for the process.
securityEvent	SecurityEventDefinition	*	ref	The reference identifies the collection of SecurityEvents that can be reported by the Process. **Stereotypes:** atpSplitable; atpUriDef **Tags:** atp.Splitkey=securityEvent atp.Status=candidate
stateDependent StartupConfig	StateDependentStartup Config	*	aggr	Applicable startup configurations.

Table A.17: Process

Class	***Referrable*** (abstract)
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).
Base	*ARObject*

▽

△

Class	**Referrable** (abstract)			
Subclasses	*AtpDefinition*, BswDistinguishedPartition, *BswModuleCallPoint*, BswModuleClientServerEntry, BswVariableAccess, CouplingPortTrafficClassAssignment, *CppImplementationDataTypeContextTarget*, *DiagnosticEnvModeElement*, EthernetPriorityRegeneration, ExclusiveAreaNestingOrder, *HwDescriptionEntity*, *ImplementationProps*, ModeTransition, *MultilanguageReferrable*, NmNetworkHandle, PncMappingIdent, *SingleLanguageReferrable*, SoConIPduIdentifier, SocketConnectionBundle, SomeipRequiredEventGroup, TimeSyncServerConfiguration, TpConnectionIdent			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Stereotypes:** atpIdentityContributor **Tags:** xml.enforceMinMultiplicity=true xml.sequenceOffset=-100
shortName Fragment	ShortNameFragment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:** xml.sequenceOffset=-90

Table A.18: Referrable

Class	**ServiceInterface**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			
Note	This represents the ability to define a PortInterface that consists of a heterogeneous collection of methods, events and fields. **Tags:** atp.recommendedPackage=ServiceInterfaces			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PortInterface*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
event	VariableDataPrototype	*	aggr	This represents the collection of events defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=event.shortName, event.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=30
field	Field	*	aggr	This represents the collection of fields defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=field.shortName, field.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=40
majorVersion	PositiveInteger	0..1	attr	Major version of the service contract. **Tags:** xml.sequenceOffset=10

▽

△

Class	**ServiceInterface**			
method	ClientServerOperation	*	aggr	This represents the collection of methods defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=method.shortName, method.variation Point.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=50
minorVersion	PositiveInteger	0..1	attr	Minor version of the service contract. **Tags:** xml.sequenceOffset=20
trigger	Trigger	*	aggr	This represents the collection of triggers defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=trigger.shortName, trigger.variation Point.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=60

Table A.19: ServiceInterface

Class	**SwComponentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Note	Role of a software component within a composition.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, CompositionSwComponentType.component			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
type	SwComponentType	0..1	tref	Type of the instance. **Stereotypes:** isOfType

Table A.20: SwComponentPrototype

Class	<<atpVariation>> **SwDataDefProps**
Package	M2::MSR::DataDictionary::DataDefProperties
Note	This class is a collection of properties relevant for data objects under various aspects. One could consider this class as a "pattern of inheritance by aggregation". The properties can be applied to all objects of all classes in which SwDataDefProps is aggregated. Note that not all of the attributes or associated elements are useful all of the time. Hence, the process definition (e.g. expressed with an OCL or a Document Control Instance MSR-DCI) has the task of implementing limitations. SwDataDefProps covers various aspects: • Structure of the data element for calibration use cases: is it a single value, a curve, or a map, but also the recordLayouts which specify how such elements are mapped/converted to the DataTypes in the programming language (or in AUTOSAR). This is mainly expressed by properties like swRecordLayout and swCalprmAxisSet • Implementation aspects, mainly expressed by swImplPolicy, swVariableAccessImplPolicy, swAddr Method, swPointerTagetProps, baseType, implementationDataType and additionalNativeTypeQualifier • Access policy for the MCD system, mainly expressed by swCalibrationAccess

▽

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
	△ • Semantics of the data element, mainly expressed by compuMethod and/or unit, dataConstr, invalid Value • Code generation policy provided by swRecordLayout **Tags:** vh.latestBindingTime=codeGenerationTime			
Base	*ARObject*			
Aggregated by	*AutosarDataType*.swDataDefProps, CompositeNetworkRepresentation.networkRepresentation, *DataPrototype*.swDataDefProps, DataPrototypeTransformationProps.networkRepresentationProps, DiagnosticDataElement.swDataDefProps, DiagnosticEnvDataElementCondition.swDataDefProps, DltArgument.networkRepresentation, FlatInstanceDescriptor.swDataDefProps, ImplementationDataTypeElement.swDataDefProps, InstantiationDataDefProps.swDataDefProps, ISignal.networkRepresentationProps, McDataInstance.resultingProperties, ParameterAccess.swDataDefProps, PerInstanceMemory.swDataDefProps, *ReceiverComSpec*.networkRepresentation, *SenderComSpec*.networkRepresentation, SomeipDataPrototypeTransformationProps.networkRepresentation, SwPointerTargetProps.swDataDefProps, SwServiceArg.swDataDefProps, SwSystemconst.swDataDefProps, SystemSignal.physicalProps			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
additionalNativeTypeQualifier	NativeDeclarationString	0..1	attr	This attribute is used to declare native qualifiers of the programming language which can neither be deduced from the baseType (e.g. because the data object describes a pointer) nor from other more abstract attributes. Examples are qualifiers like "volatile", "strict" or "enum" of the C-language. All such declarations have to be put into one string. **Tags:** xml.sequenceOffset=235
annotation	Annotation	*	aggr	This aggregation allows to add annotations (yellow pads ...) related to the current data object. **Tags:** xml.roleElement=true xml.roleWrapperElement=true xml.sequenceOffset=20 xml.typeElement=false xml.typeWrapperElement=false
baseType	SwBaseType	0..1	ref	Base type associated with the containing data object. **Tags:** xml.sequenceOffset=50
compuMethod	CompuMethod	0..1	ref	Computation method associated with the semantics of this data object. **Tags:** xml.sequenceOffset=180
dataConstr	DataConstr	0..1	ref	Data constraint for this data object. **Tags:** xml.sequenceOffset=190
displayFormat	DisplayFormatString	0..1	attr	This property describes how a number is to be rendered e.g. in documents or in a measurement and calibration system. **Tags:** xml.sequenceOffset=210
displayPresentation	DisplayPresentationEnum	0..1	attr	This attribute controls the presentation of the related data for measurement and calibration tools.

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
implementation DataType	AbstractImplementation DataType	0..1	ref	This association denotes the ImplementationDataType of a data declaration via its aggregated SwDataDefProps. It is used whenever a data declaration is not directly referring to a base type. Especially • redefinition of an ImplementationDataType via a "typedef" to another ImplementationDatatype • the target type of a pointer (see SwPointerTarget Props), if it does not refer to a base type directly • the data type of an array or record element within an ImplementationDataType, if it does not refer to a base type directly • the data type of an SwServiceArg, if it does not refer to a base type directly **Tags:** xml.sequenceOffset=215
invalidValue	ValueSpecification	0..1	aggr	Optional value to express invalidity of the actual data element. **Tags:** xml.sequenceOffset=255
stepSize	Float	0..1	attr	This attribute can be used to define a value which is added to or subtracted from the value of a DataPrototype when using up/down keys while calibrating.
swAddrMethod	SwAddrMethod	0..1	ref	Addressing method related to this data object. Via an association to the same SwAddrMethod it can be specified that several DataPrototypes shall be located in the same memory without already specifying the memory section itself. **Tags:** xml.sequenceOffset=30
swAlignment	AlignmentType	0..1	attr	The attribute describes the intended typical alignment of the DataPrototype. If the attribute is not defined the alignment is determined by the swBaseType size and the memoryAllocationKeywordPolicy of the referenced Sw AddrMethod. **Tags:** xml.sequenceOffset=33
swBit Representation	SwBitRepresentation	0..1	aggr	Description of the binary representation in case of a bit variable. **Tags:** xml.sequenceOffset=60
swCalibration Access	SwCalibrationAccess Enum	0..1	attr	Specifies the read or write access by MCD tools for this data object. **Tags:** xml.sequenceOffset=70
swCalprmAxis Set	SwCalprmAxisSet	0..1	aggr	This specifies the properties of the axes in case of a curve or map etc. This is mainly applicable to calibration parameters. **Tags:** xml.sequenceOffset=90
swComparison Variable	SwVariableRefProxy	*	aggr	Variables used for comparison in an MCD process. **Tags:** xml.sequenceOffset=170 xml.typeElement=false
swData Dependency	SwDataDependency	0..1	aggr	Describes how the value of the data object has to be calculated from the value of another data object (by the MCD system). **Tags:** xml.sequenceOffset=200

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
swHostVariable	SwVariableRefProxy	0..1	aggr	Contains a reference to a variable which serves as a host-variable for a bit variable. Only applicable to bit objects. **Tags:** xml.sequenceOffset=220 xml.typeElement=false
swImplPolicy	SwImplPolicyEnum	0..1	attr	Implementation policy for this data object. **Tags:** xml.sequenceOffset=230
swIntended Resolution	Numerical	0..1	attr	The purpose of this element is to describe the requested quantization of data objects early on in the design process. The resolution ultimately occurs via the conversion formula present (compuMethod), which specifies the transition from the physical world to the standardized world (and vice-versa) (here, "the slope per bit" is present implicitly in the conversion formula). In the case of a development phase without a fixed conversion formula, a pre-specification can occur through swIntendedResolution. The resolution is specified in the physical domain according to the property "unit". **Tags:** xml.sequenceOffset=240
swInterpolation Method	Identifier	0..1	attr	This is a keyword identifying the mathematical method to be applied for interpolation. The keyword needs to be related to the interpolation routine which needs to be invoked. **Tags:** xml.sequenceOffset=250
swIsVirtual	Boolean	0..1	attr	This element distinguishes virtual objects. Virtual objects do not appear in the memory, their derivation is much more dependent on other objects and hence they shall have a swDataDependency . **Tags:** xml.sequenceOffset=260
swPointerTarget Props	SwPointerTargetProps	0..1	aggr	Specifies that the containing data object is a pointer to another data object. Note: This atpSplitable property has no atp.Splitkey due to atpVariation (PropertySetPattern). **Stereotypes:** atpSplitable **Tags:** xml.sequenceOffset=280
swRecord Layout	SwRecordLayout	0..1	ref	Record layout for this data object. **Tags:** xml.sequenceOffset=290
swRefresh Timing	MultidimensionalTime	0..1	aggr	This element specifies the frequency in which the object involved shall be or is called or calculated. This timing can be collected from the task in which write access processes to the variable run. But this cannot be done by the MCD system. So this attribute can be used in an early phase to express the desired refresh timing and later on to specify the real refresh timing. **Tags:** xml.sequenceOffset=300
swTextProps	SwTextProps	0..1	aggr	the specific properties if the data object is a text object. **Tags:** xml.sequenceOffset=120

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
swValueBlock Size	Numerical	0..1	attr	This represents the size of a Value Block **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime xml.sequenceOffset=80
swValueBlock SizeMult (ordered)	Numerical	*	attr	This attribute is used to specify the dimensions of a value block (VAL_BLK) for the case that that value block has more than one dimension. The dimensions given in this attribute are ordered such that the first entry represents the first dimension, the second entry represents the second dimension, and so on. For one-dimensional value blocks the attribute swValue BlockSize shall be used and this attribute shall not exist. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime
unit	Unit	0..1	ref	Physical unit associated with the semantics of this data object. This attribute applies if no compuMethod is specified. If both units (this as well as via compuMethod) are specified the units shall be compatible. **Tags:** xml.sequenceOffset=350
valueAxisData Type	ApplicationPrimitive DataType	0..1	ref	The referenced ApplicationPrimitiveDataType represents the primitive data type of the value axis within a compound primitive (e.g. curve, map). It supersedes CompuMethod, Unit, and BaseType. **Tags:** xml.sequenceOffset=355

Table A.21: SwDataDefProps

Class	**SymbolProps**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	This meta-class represents the ability to contribute a part of a namespace.			
Base	*ARObject*, *ImplementationProps*, *Referrable*			
Aggregated by	Allocator.namespace, ApApplicationErrorDomain.namespace, *AtomicSwComponentType*.symbolProps, *CppImplementationDataType*.namespace, ImplementationDataType.symbolProps, *PortInterface*.namespace, SecurityEventDefinition.eventSymbolName			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.22: SymbolProps

B Platform Extension API (normative)

The focus of the APIs in this section are for OEM-specific platform extensions. The abstraction of the interfaces is lower which could lead to a higher machine dependency

B.1 Tracing Interface in ara::log

The tracing interface is not for applications. It is internally used in ara::log to provide an interface for external trace tools.

B.1.1 TraceArti

[SWS_LOG_20000]{DRAFT} **Definition of API function ara::log::ext::TraceArti** ⌈

Kind:	function	
Header file:	#include "ara/log/trace_arti.h"	
Scope:	namespace ara::log::ext	
Symbol:	TraceArti(const MsgId &msg_id, const Params &... params)	

Syntax:	`template <typename MsgId, typename... Params>`
	`void TraceArti (const MsgId &msg_id, const Params &... params)`
	`noexcept;`	

Template param:	MsgId	the type of the msg_id parameter
	Params	the types of the params parameters
Parameters (in):	msg_id	an implementation-defined type identifying the message object
	params	the arguments of the message
Return value:	None	
Exception Safety:	noexcept	
Description:	Handles a modeled message, specific to the used trace tool. This function is only called internally by the Logger framework when switching between Log and TraceArti. It represents the interface to trace tools. The runtime overhead of this function shall be kept as minimal as possible, e.g. by using specializations based on the MsgId.	

⌋*(RS_LT_00003)*

C Change History

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

C.1 Change History of this document according to AUTOSAR Release R23-11

C.1.1 Added Specification Items in R23-11

Number	Heading
[SWS_LOG_00172]	Definition of API class ara::log::Logger
[SWS_LOG_20000]	Definition of API function ara::log::ext::TraceArti
[SWS_LOG_20001]	
[SWS_LOG_20002]	
[SWS_LOG_20003]	
[SWS_LOG_20004]	
[SWS_LOG_20005]	

Table C.1: Added Specification Items in R23-11

C.1.2 Changed Specification Items in R23-11

Number	Heading
[SWS_LOG_00005]	
[SWS_LOG_00018]	Definition of API enum ara::log::LogLevel
[SWS_LOG_00021]	Definition of API function ara::log::CreateLogger
[SWS_LOG_00039]	Definition of API function ara::log::LogStream::Flush
[SWS_LOG_00040]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00041]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00042]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00043]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00044]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00045]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00046]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00047]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00048]	Definition of API function ara::log::LogStream::operator<<

▽

△

Number	Heading
[SWS_LOG_00049]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00050]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00051]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00062]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00063]	Definition of API function ara::log::operator<<
[SWS_LOG_00064]	Definition of API function ara::log::Logger::LogFatal
[SWS_LOG_00065]	Definition of API function ara::log::Logger::LogError
[SWS_LOG_00066]	Definition of API function ara::log::Logger::LogWarn
[SWS_LOG_00067]	Definition of API function ara::log::Logger::LogInfo
[SWS_LOG_00068]	Definition of API function ara::log::Logger::LogDebug
[SWS_LOG_00069]	Definition of API function ara::log::Logger::LogVerbose
[SWS_LOG_00070]	Definition of API function ara::log::Logger::IsEnabled
[SWS_LOG_00098]	Definition of API enum ara::log::ClientState
[SWS_LOG_00124]	Definition of API function ara::log::operator<<
[SWS_LOG_00125]	Definition of API function ara::log::operator<<
[SWS_LOG_00126]	Definition of API function ara::log::operator<<
[SWS_LOG_00127]	Definition of API function ara::log::operator<<
[SWS_LOG_00128]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00129]	Definition of API function ara::log::LogStream::WithLocation
[SWS_LOG_00131]	Definition of API function ara::log::Logger::WithLevel
[SWS_LOG_00132]	Definition of API function ara::log::LogStream::WithTag
[SWS_LOG_00133]	Definition of API function ara::log::Logger::LogWith
[SWS_LOG_00173]	Definition of API class ara::log::LogStream
[SWS_LOG_00201]	Definition of API function ara::log::Arg
[SWS_LOG_00203]	Definition of API function ara::log::LogStream::operator<<
[SWS_LOG_00204]	Definition of API function ara::log::Logger::Log
[SWS_LOG_00205]	Definition of API function ara::log::RegisterConnectionStateHandler
[SWS_LOG_00206]	Definition of API enum ara::log::Fmt
[SWS_LOG_00207]	Definition of API class ara::log::Format
[SWS_LOG_00208]	Definition of API function ara::log::Dflt
[SWS_LOG_00209]	Definition of API function ara::log::Hex
[SWS_LOG_00210]	Definition of API function ara::log::Hex
[SWS_LOG_00211]	Definition of API function ara::log::Dec
[SWS_LOG_00212]	Definition of API function ara::log::Dec
[SWS_LOG_00213]	Definition of API function ara::log::Oct
[SWS_LOG_00214]	Definition of API function ara::log::Oct
[SWS_LOG_00215]	Definition of API function ara::log::Bin
[SWS_LOG_00216]	Definition of API function ara::log::Bin

▽

△

Number	Heading
[SWS_LOG_00217]	Definition of API function ara::log::DecFloat
[SWS_LOG_00218]	Definition of API function ara::log::DecFloatMax
[SWS_LOG_00219]	Definition of API function ara::log::EngFloat
[SWS_LOG_00220]	Definition of API function ara::log::EngFloatMax
[SWS_LOG_00221]	Definition of API function ara::log::HexFloat
[SWS_LOG_00222]	Definition of API function ara::log::HexFloatMax
[SWS_LOG_00223]	Definition of API function ara::log::AutoFloat
[SWS_LOG_00224]	Definition of API function ara::log::AutoFloatMax
[SWS_LOG_00225]	Definition of API variable ara::log::Format::fmt
[SWS_LOG_00226]	Definition of API variable ara::log::Format::precision
[SWS_LOG_00255]	Definition of API function ara::log::Logger::SetThreshold
[SWS_LOG_00256]	Definition of API function ara::log::CreateLogger
[SWS_LOG_00258]	Definition of API function ara::log::LogStream::WithPrivacy
[SWS_LOG_00261]	Definition of API class ara::log::Argument
[SWS_LOG_00263]	Definition of API function ara::log::CreateLogger

Table C.2: Changed Specification Items in R23-11

C.1.3 Deleted Specification Items in R23-11

none