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

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-03

Document Change History

Date	Release	Changed by	Description
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

Table of Contents

1 Introduction and functional overview

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

The `Log and Trace` functional cluster provides interfaces for applications to forward logging information onto the communication bus, the console, or to the file system. 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 DLT protocol can be used which is standardized within the AUTOSAR consortium.

The DLT protocol can add additional information, like an ECU ID, to the provided logging information. This information can be used by a DLT Logging Client to relate, sort or filter the received logging frames.

Detailed information regarding the use cases and the DLT protocol itself are provided by the PRS DLT protocol specification[1].

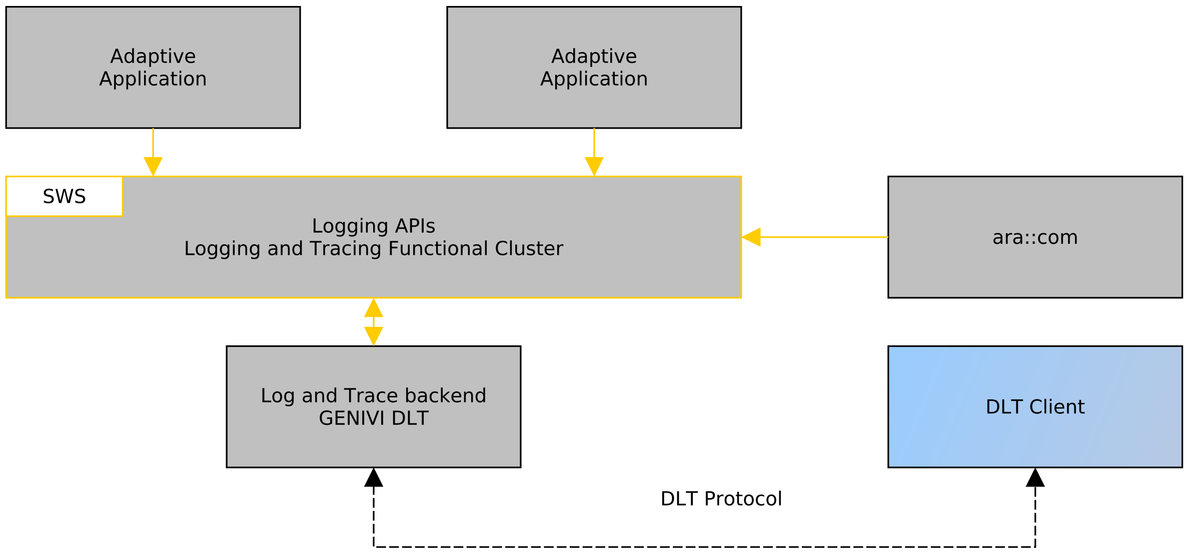

Figure 1.1: Architecture overview

Furthermore, this document introduces additional specification extensions for the `Adaptive Platform Log and Trace` functional cluster. Such extensions consist of several requirements:

- **[SWS_LOG_00080]** ⌈ Log and Trace information shall be able to be correlated across time and between multiple ECUs. ⌋*()*

- **[SWS_LOG_00081]** ⌈ Communication flow between different`Adaptive Applications` shall be traceable, regardless of the scope of the communication (i.e. internal or external communication). ⌋*()*

2 Acronyms and Abbreviations

Abbreviation / Acronym:	Description:
L&T	Log and Trace
DLT protocol	Original name of the protocol itself (Diagnostic Log and Trace)
Logging API	The main logging interface towards user applications as a library
Logging back-end	Implementation of the Logging 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
Log message	Log message, including header(s)
Log severity level	Meta information about the severity of a passed logging information
PoD	Plain old data type supported natively by most platforms. Integers, floats, chars, etc.

3 Related documentation

3.1 Input documents

[1] Log and Trace Protocol Specification
AUTOSAR_PRS_LogAndTraceProtocol

[2] Requirements on Log and Trace
AUTOSAR_RS_LogAndTrace

[3] Specification of Time Synchronization for Adaptive Platform
AUTOSAR_SWS_TimeSync

[4] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

4 Constraints and assumptions

4.1 Limitations

The provided Logging framework API is designed to be independent from the underlying logging protocol 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

There are no dependencies to other functional clusters.

5.1 Platform dependencies

This specification is part of the AUTOSAR Adaptive Platform and therefore depends on it.

6 Requirements Tracing

The following table references the requirements specified in RS Log And Trace [2] 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_00003]	Applications shall have the possibility to send log or trace messages to the LT module.	[SWS_LOG_00002]
[RS_LT_00044]	Provide raw buffer content.	[SWS_LOG_00014] [SWS_LOG_00038] [SWS_LOG_00061]
[RS_LT_00045]	Check the current severity level.	[SWS_LOG_00007] [SWS_LOG_00070]
[RS_LT_00046]	Conversion functions for hexadecimal and binary values.	[SWS_LOG_00015] [SWS_LOG_00016] [SWS_LOG_00017] [SWS_LOG_00022] [SWS_LOG_00023] [SWS_LOG_00024] [SWS_LOG_00025] [SWS_LOG_00026] [SWS_LOG_00027] [SWS_LOG_00028] [SWS_LOG_00029] [SWS_LOG_00030] [SWS_LOG_00031] [SWS_LOG_00032] [SWS_LOG_00033] [SWS_LOG_00034] [SWS_LOG_00035] [SWS_LOG_00036] [SWS_LOG_00037] [SWS_LOG_00053] [SWS_LOG_00054] [SWS_LOG_00055] [SWS_LOG_00056] [SWS_LOG_00057] [SWS_LOG_00058] [SWS_LOG_00059] [SWS_LOG_00060]
[RS_LT_00047]	Initialization and registration.	[SWS_LOG_00003] [SWS_LOG_00004] [SWS_LOG_00020]
[RS_LT_00048]	Meta information about Applications.	[SWS_LOG_00004] [SWS_LOG_00020]
[RS_LT_00049]	Providing Logging Information.	[SWS_LOG_00008] [SWS_LOG_00009] [SWS_LOG_00010] [SWS_LOG_00011] [SWS_LOG_00012] [SWS_LOG_00013] [SWS_LOG_00018] [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]
[RS_LT_00050]	Grouping of Logging Information.	[SWS_LOG_00005] [SWS_LOG_00006] [SWS_LOG_00021]
[RS_LT_00051]	Logging Information targets.	[SWS_LOG_00019]
[RS_LT_00052]	Early logging.	[SWS_LOG_00001]

7 Functional specification

This specification defines the usage of the defined C++ 11 APIs for the functional cluster Log & Trace. Applications can use these functions to forward logging information to the bus, the console or the file system.

The following functionalities are provided:

1) Methods for initializing the Logging framework (see chapter 7.2)
2) Utility methods to convert decimal values into hexadecimal or binary values (see chapter 7.3)
3) Automatic timestamping of log messages (see chapter 7.4)
4) Tracing of communication between applications (see chapter 7.5)
5) Network load balancing (see chapter 7.6)

7.1 Necessary Parameters and Initialization

The concept of identifying the user application:
To be able to distinguish the logs from different applications with in a system (e.g. an ECU or even the whole vehicle), every application, 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, for every context within an application a particular ID and a description has to be assigned.

Every application can have an arbitrary amount of contexts, but at least one – the default context.

The application using the Log & Trace framework needs to configure it once at early startup with the following information:

- Application ID
- Application description
- The default log level
- The log mode
- The log file path

The application using the Logging framework can request contexts from it by providing the following information:

- Context ID

- Context description

7.1.1 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, 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 the vehicle or at least within one ECU.

Note:
It is also recommended to assign unique IDs per application process, meaning if the same application is started multiple times it shall have an own ID per instance.

7.1.2 Application Description

Since the length of the Application ID can be quite short and an additional descriptive text needs to be provided. This Application description is passed as a string. The maximum length is implementation dependent.

7.1.3 Default Log Level

The Log Level represents the severity of the log messages. Severity levels are defined in chapter 7.2.

Each initiated log message is qualified with such a severity level.
The default Log Level is set at initialization per Application ID.
The application log level acts as a reporting filter. Only log messages having a higher or the same severity will be processed by the Logging framework, the rest is ignored.

The Default Log Level is the initial configured log reporting level for a certain application.

The application wide log reporting level shall be adjustable during runtime. The realization is implementation detail of the underlying back-end. E.g. remotely, via a client (DLT). Same applies for the context reporting level.

Design rationale of providing a initial default log reporting level only application wide vs. having them per context level:

- Simplifies the API usage. Otherwise the user will have to define a context level for each group before using the API.
- The context separation of log messages will be still possible during runtime.

7.1.4 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, saved into a file on the file system, or sent over the communication bus. Any combination of these destinations can also be simultaneously selected.

7.1.5 Log File Path

In case the file system mode is set as a destination directory path needs to be provided. The actual file name will be generated by the Logging framework.

7.1.6 Context ID

The Context ID is an identifier that is used to logically group logging information within the scope of a user application. The Context ID is passed as a string value. Depending on the actual implementation of the Logging framework, the length of the Context ID might be limited. To be able to unambiguously associate the received logging information, it is recommended to assign unique Context IDs within each Application.

Note:
Special attention should be paid to library components, that are meant to be used by applications and therefore are running within the application's process scope. Logs done out of those libraries will end-up inside the scope of the parent application. In order to distinguish the internal library logs from the application 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.

7.1.7 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.

7.1.8 Initialization of the Logging framework

Before the logging information can be processed, the Logging framework needs to be initialized. In order to initialize the Logging framework, the application needs to provide the mandatory information to the Logging framework.

The Application ID is used to identify and to associate the provided logging information, whereas the log mode defines where the logging information is routed. Possible destinations are the console, the file system, or the communication bus. These three destinations can also be used simultaneously combined.

Next to the registration of the applications at the Logging framework, contexts need to be registered as well. These contexts are used to logically cluster logging information.

In addition to registration of the applications and the contexts, applications can query the current active severity.

[SWS_LOG_00001] ⌈ All messages logged before the initialization of the Logging framework is done shall be stored inside a FIFO-buffer with a limited size. That means, oldest entries are lost if the buffer exceeds. The size of the buffer is implementation detail. ⌋*(RS_LT_00052)*

[SWS_LOG_00002] ⌈ In case of any errors occurring inside the Logging framework or underlying system, it is intended to not bother the applications and silently discard the function calls. For this purpose, the relevant interfaces (see chapter 8) neither specify return values nor throw exceptions. ⌋*(RS_LT_00003)*

[SWS_LOG_00003] ⌈ Before log messages can be processed, the `InitLogging()` function needs to be called. This function initializes the Logging framework for the application with the given properties. ⌋*(RS_LT_00047)*

[SWS_LOG_00004] ⌈ By calling `InitLogging()`, the following parameters need to be provided:

- Application ID
- Application description
- The default log level
- The log mode
- The directory path (only necessary if LogMode::kFile is given as log mode)

⌋*(RS_LT_00047, RS_LT_00048)*

Note:
Depending on the Logging framework implementation not all of the features might be supported, hence not all of the properties will be used.

[SWS_LOG_00005] ⌈ Before log message can be processed, at least one logger context has to be available. Calling function `CreateLogger()` will create a logger context instance internally inside the Logging framework and returned as reference to the using application. 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 protocol back-end, its lifetime must be ensured until the end of the application's process. ⌋*(RS_LT_00050)*

[SWS_LOG_00006] ⌈ By calling `CreateLogger()`, the following parameters need to be provided:

- The context ID
- The context description

⌋*(RS_LT_00050)*

[SWS_LOG_00007] ⌈ Applications should be able to check if a desired log level is configured through the function `IsLogEnabled()`. 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_00045)*

7.2 Log Messages

The Adaptive Logging framework offers stream based API for message creation that supports certain data types described below.

Design rationale for having insert stream based API vs. function-like solutions:

- Convenient usage for developers
- De-facto standard way of concatenating args in C++ or in other words, passing data to objects
- Enables easy way of having a multi-line message builder

Performance remark:
C++ stream operators translates to normal function calls after compilation, it is just another syntax, there is no difference compared to functions having a variadic argument pack. Actually compilers expand them in the same way.

To forward log messages to the Logging framework, C++ interfaces are provided. For every severity (also known as log level), a separate function call is foreseen.

The following severity levels are defined:

- Off (Logging data is turned off)
- Fatal (Fatal system errors)
- Error (Error messages with impact on correct functionality)
- Warn (Warning messages if correct behavior cannot be ensured)
- Info (Informational log messages providing high level understanding of the program flow)
- Debug (Detailed debug used during development)
- Verbose (Verbose debug information used during development)

Note:
Off is not applicable for log message. This level can be used to set reporting level for the Logging framework either initially in InitLogging() or during runtime.

Design Rationale:
For having separate functions per log level vs. passing log level as parameter to a generic function:

- Convenient usage of the API, less to type, clearer reading
- Technically no difference, just a shortcut

Each of the log message is represented as a stream object which is an instance of the LogStream class.

By calling one of the Log*() functions, a temporary unnamed LogStream object will be created with a scoped life time, that lasts until the end of the statement.

Design rationale for having temporary stream objects vs. some global-buffer-based log solution (e.g. std::cout):

- Required **destructor** semantic to express **end-of-statement**
- End-of-statement expression is required to gain **scoped** resource **access**
- Guaranteed scoped access if required to ensure **thread safety** which enables to log out messages concurrently and have them processed in one piece
- Convenient usage for developer due to the fact that he does not need to care for resource-life-cycle (the stream object goes automatically out-of-scope)

Performance remark:

- Costs of constructor/destructor depends on their content and is implementation detail of the Logging framework.
- Costs of trivial constructor and destructor (e.g. empty ones) is cheap, actually instantiating an object in C++ equals to instantiating a struct in C.
- Logger class API is designed to create a stack object of LogStream and passes them back via RVO (return-value-optimization is C++11 ISO standard), which results in a no-cost operation for the transition of a LogStream object after a Log*() function call.

Store LogStream objects in a variable:
It is also possible to use the API in an alternative way by storing a 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 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 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 API is to get the multi-line builder functionality.

Note:
It is highly advised NOT to hold global LogStream objects in multi-threaded applications,because then concurrent access protection will no longer be covered by the Logging API." that is simply the fact, who needs to do what because of that fact is not our concern.

Usage examples:

```
// unnamed temporary LogStream object will process
// the arguments and dies after ";"
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
LogStream logInfo = LogInfo();
logInfo ≪ "some log information" ≪ 123;
logInfo ≪ "some other information";
logInfo.Flush();
logInfo ≪ "a new message..." ≪ 456;
```

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

[SWS_LOG_00008] ⌈ To initiate a log message with the Log level `Fatal`, the API `LogFatal()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00009] ⌈ To initiate a log message with the Log level `Error`, the API `LogError()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00010] ⌈ To initiate a log message with the Log level `Warning`, the API `LogWarn()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00011] ⌈ To initiate a log message with the Log level `Info`, the API `LogInfo()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00012] ⌈ To initiate a log message with the Log level `Debug`, the API `LogDebug()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00013] ⌈ To initiate a log message with the Log level `Verbose`, the API `LogVerbose()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator "≪". ⌋*(RS_LT_00049)*

[SWS_LOG_00014] ⌈ To log raw data by providing a buffer, the API `RawBuffer()` shall be called. ⌋*(RS_LT_00044)*

7.3 Conversion functions

Sometimes it makes sense to represent integer numbers in hexadecimal or binary format instead of decimal format.
For this purpose, the following functions are defined to convert provided decimal numbers into the hexadecimal or binary system.

[SWS_LOG_00015] ⌈ In case a decimal number is converted into a string with hexadecimal or binary representation, the most significant bit shall be set to '1' for negative numbers and to '0' for positive numbers. ⌋*(RS_LT_00046)*

[SWS_LOG_00016] ⌈ Function `HexFormat()` shall provide functionality to convert an integer decimal number into a string with hexadecimal representation. ⌋*(RS_LT_00046)*

[SWS_LOG_00017] ⌈ Function `BinFormat()` shall provide functionality to convert an integer decimal number into a string with binary representation. ⌋*(RS_LT_00046)*

7.4 Log and Trace timestamp

The Log and Trace information is transmitted by means of the Log and Trace 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 [2] for more information). The synchronized time base is supplied by the Time Synchronization functional cluster (refer to [3] for more information).

[SWS_LOG_00082] ⌈ Log and Trace shall have accesss to a synchronized time base. ⌋*()*

Note:
Which time base resource is going to be used, to access the time information, depends on the manifest configuration.

[SWS_LOG_00083] ⌈ In case there is no time base resource referenced by the Log and Trace module in the manifest configuration, no timestamp information shall be transmitted. ⌋*()*

[SWS_LOG_00091] ⌈ Each time the trace feature is enabled , Log and Trace shall send a message indicating whether the used time base is a local time base or a globally synchronized time base. ⌋*()*

[SWS_LOG_00092] ⌈ If the referenced time base changes, Log and Trace shall provide a trace message informing about this change. ⌋*()*

[SWS_LOG_00093] ⌈ If the referenced time base:

- is a globally synchronized time base
- loses synchronicity (i.e. there is an interruption on the network communication)

Log and Trace shall inform via a trace message of such loss of synchronicity. ⌋*()*

[SWS_LOG_00094] ⌈ If the referenced time base:

- is a globally synchronized time base
- it is updated presenting a leap jump (either to the future or to the past)

Log and Trace shall inform via a trace message that the time base has been updated and it shall provide the delta value (i.e. the difference between the updated time base and the previous time base). A signed data type shall be used to indicate if the leap jump has been done into the past (a negative value) or into the future (positive value). ⌋*()*

Note:
At the moment there is no standardized format for the trace messages. Therefore, it

should be considered that there are implementation specific messages.

7.5 Application Communication tracing

Tracing of the Communication between `Adaptive Applications` has paramount advantage when analyzing information flow for different reasons, from debugging to measuring communication latencies to profiling different communication events.

[SWS_LOG_00084] ⌈ Communication tracing of an `Adaptive Application` shall be configurable by means of the manifest configuration. ⌋*()*

[SWS_LOG_00085] ⌈ If application communication tracing is enabled, it shall be possible to start or to stop the trace of a specific port of an `Adaptive Application` during runtime. ⌋*()*

[SWS_LOG_00096] ⌈ If the communication tracing of an `Adaptive Application` is enabled (in the manifest configuration), the Context Identification of each port of such `Adaptive Application` shall be assigned. ⌋*()*

[SWS_LOG_00086] ⌈ The application communication tracing of an `Adaptive Application` shall be done without any interaction and independent from the `Adaptive Application`. ⌋*()*

[SWS_LOG_00087] ⌈ The application communication tracing of an `Adaptive Applications` shall be done within the scope of the Communication Ports of such applications. ⌋*()*

[SWS_LOG_00088] ⌈ The trace (and log) information must be serialized using a self-describing format (i.e. JSON). ⌋*()*

7.6 Log and Trace message network load balancing

[SWS_LOG_00090] ⌈ The bandwith consumption, effectively the speed at which the Log and Trace information is being sent on the network bus shall not be higher than 60 percent of the total possible bandwidth of the network bus. ⌋*()*

[SWS_LOG_00095] ⌈ When Log and Trace receives a high load of trace information (generated at the same time) from multiple `Adaptive Applications`, it shall buffer this data internally so it can be sent continuously and so that no information is lost. ⌋*()*

8 API specification

8.1 Type definitions

8.1.1 LogLevel

[SWS_LOG_00018] ⌈ Type `LogLevel` shall be defined as described in table 8.1. ⌋
(RS_LT_00049)

Name:	LogLevel		
Type:	uint8_ t		
Range:	kOff	0	No logging.
	kFatal	1	Fatal error.
	kError	2	Error with impact to correct functionality.
	kWarn	3	Warning if correct behavior cannot be ensured.
	kInfo	4	Informational, high level understanding.
	kDebug	5	Detailed information for programmers.
	kVerbose	6	Extra-verbose debug messages.

Syntax:	`enum class LogLevel : uint8_t {`
	`    kOff,`
	`    kFatal,`
	`    kError,`
	`    kWarn,`
	`    kInfo,`
	`    kDebug,`
	`    kVerbose`
	`};`		

Header file:	ara/log/common.h		
Description:	List of possible severity levels.		

Table 8.1: Type definition - LogLevel

8.1.2 LogMode

[SWS_LOG_00019] ⌈ Type `LogMode` shall be defined as described in table 8.2. ⌋
(RS_LT_00051)

Name:	LogMode		
Type:	uint8_ t		
Range:	kRemote	0x01	Sent remotely.
	kFile	0x02	Save to file.

	kConsole	0x04	Forward to console.

Syntax:	`enum class LogMode : uint8_t {`
	`kRemote,`
	`kFile,`
	`kConsole`
	`};`		

Header file:	ara/log/common.h		
Description:	Log mode. Flags, used to configure the sink for log messages. Note: In order to combine flags, at least the OR and AND operators needs to be provided for this type.		

Table 8.2: Type definition - LogMode

8.1.3 LogHex8

Name	LogHex8
Kind	Type
Derived from	uint8_t
Description	Represents a 8 bit hexadecimal value data type

Table 8.3: Definition of LogHex8

8.1.4 LogHex16

Name	LogHex16
Kind	Type
Derived from	uint16_ t
Description	Represents a 16 bit hexadecimal value data type

Table 8.4: Definition of LogHex16

8.1.5 LogHex32

Name	LogHex32
Kind	Type
Derived from	uint32_ t
Description	Represents a 32 bit hexadecimal value data type

Table 8.5: Definition of LogHex32

8.1.6 LogHex64

Name	LogHex64
Kind	Type

Derived from	uint64_ t
Description	Represents a 64 bit hexadecimal value data type

Table 8.6: Definition of LogHex64

8.1.7 LogBin8

Name	LogBin8
Kind	Type
Derived from	uint8_ t
Description	Represents a 8 bit binary data type

Table 8.7: Definition of LogBin8

8.1.8 LogBin16

Name	LogBin16
Kind	Type
Derived from	uint16_ t
Description	Represents a 16 bit binary data type

Table 8.8: Definition of LogBin16

8.1.9 LogBin32

Name	LogBin32
Kind	Type
Derived from	uint32_ t
Description	Represents a 32 bit binary data type

Table 8.9: Definition of LogBin32

8.1.10 LogBin64

Name	LogBin64
Kind	Type
Derived from	uint64_ t
Description	Represents a 64 bit binary data type

Table 8.10: Definition of LogBin64

8.2 Function definitions

8.2.1 InitLogging

[SWS_LOG_00020] ⌈ Method `InitLogging` shall be defined as described in table 8.11. ⌋*(RS_LT_00047, RS_LT_00048)*

Service name:	InitLogging	

Syntax:	`void InitLogging(`
	`std::string appId,`
	`std::string appDescription,`
	`LogLevel appDefLogLevel,`
	`LogMode logMode,`
	`std::string directoryPath`
	`) noexcept;`	

Parameters (in):	appId	The ID of the Application
	appDescription	Description of the Application
	appDefLogLevel	The application's default log level
	logMode	The log mode(s) to be used
	directoryPath	The directory path for the file log mode
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Exceptions:	None	
Header file:	ara/log/logging.h	

Description:	Initializes the logging framework for the application with given properties. In case the kFile flag is set in logMode, the directory path needs to be provided. The actual file name will be generated by the Logging framework.
	
	Note: The call to InitLogging shall be done as early as possible inside the program runnable (e.g. the main() function or some init function).
	
	Usage:
	int main(int argc, char* argv[])
	{
	InitLogging("ABCD", "This is the application known as ABCD", LogLevel::kVerbose, LogMode::kRemote);
	}	

Table 8.11: Method definition - InitLogging

8.2.2 CreateLogger

[SWS_LOG_00021] ⌈ Method `CreateLogger` shall be defined as described in table 8.12. ⌋*(RS_LT_00050)*

Service name:	CreateLogger	
Syntax:	`Logger& CreateLogger( std::string ctxId, std::string ctxDescription) noexcept;`	
Parameters (in):	ctxId	The context ID

	ctxDescription	The description of the provided context ID
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Reference to the internal managed instance of a Logger object. Ownership stays within the Logging framework.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Creates a Logger object, holding the context which is registered in the Logging framework.	

Table 8.12: Method definition - CreateLogger

8.2.3 HexFormat (uint8)

[SWS_LOG_00022] conversion of a uint8 into a hexadecimal value ⌈ Method `HexFormat (uint8)` shall be defined as described in table 8.13. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex8 HexFormat(uint8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex8 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format.	

Table 8.13: Method definition - HexFormat (uint8)

8.2.4 HexFormat (int8)

[SWS_LOG_00023] conversion of an int8 into a hexadecimal value ⌈ Method `HexFormat (int8)` shall be defined as described in table 8.14. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex8 HexFormat(int8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex8 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

Table 8.14: Method definition - HexFormat (int8)

8.2.5 HexFormat (uint16)

[SWS_LOG_00024] conversion of a uint16 into a hexadecimal value ⌈ Method `HexFormat (uint16)` shall be defined as described in table 8.15. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex16 HexFormat(uint16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex16 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format.	

Table 8.15: Method definition - HexFormat (uint16)

8.2.6 HexFormat (int16)

[SWS_LOG_00025] conversion of an int16 into a hexadecimal value ⌈ Method `HexFormat (int16)` shall be defined as described in table 8.16. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex16 HexFormat(int16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex16 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

Table 8.16: Method definition - HexFormat (int16)

8.2.7 HexFormat (uint32)

[SWS_LOG_00026] conversion of a uint32 into a hexadecimal value ⌈ Method `HexFormat (uint32)` shall be defined as described in table 8.17. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex32 HexFormat(uint32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex32 type that has a built-in stream handler.	
Exceptions:	None	

Header file:	ara/log/logging.h
Description:	Logs decimal numbers in hexadecimal format.

Table 8.17: Method definition - HexFormat (uint32)

8.2.8 HexFormat (int32)

[SWS_LOG_00027] conversion of an int32 into a hexadecimal value ⌈ Method `HexFormat (int32)` shall be defined as described in table 8.18. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex32 HexFormat(int32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex32 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

Table 8.18: Method definition - HexFormat (int32)

8.2.9 HexFormat (uint64)

[SWS_LOG_00028] conversion of a uint64 into a hexadecimal value ⌈ Method `HexFormat (uint64)` shall be defined as described in table 8.19. ⌋*(RS_LT_00046)*

Service name:	HexFormat	
Syntax:	`LogHex64 HexFormat(uint64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex64 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format.	

Table 8.19: Method definition - HexFormat (uint64)

8.2.10 HexFormat (int64)

[SWS_LOG_00029] conversion of an int64 into a hexadecimal value ⌈ Method `HexFormat (int64)` shall be defined as described in table 8.20. ⌋*(RS_LT_00046)*

Service name:	HexFormat

Syntax:	`LogHex64 HexFormat(int64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogHex64 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

Table 8.20: Method definition - HexFormat (int64)

8.2.11 BinFormat (uint8)

[SWS_LOG_00030] conversion of a uint8 into a binary value ⌈ Method `BinFormat (uint8)` shall be defined as described in table 8.21. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin8 BinFormat(uint8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin8 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format.	

Table 8.21: Method definition - BinFormat (uint8)

8.2.12 BinFormat (int8)

[SWS_LOG_00031] conversion of an int8 into a binary value ⌈ Method `BinFormat (int8)` shall be defined as described in table 8.22. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin8 BinFormat(int8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin8 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

Table 8.22: Method definition - BinFormat (int8)

8.2.13 BinFormat (uint16)

[SWS_LOG_00032] conversion of a uint16 into a binary value ⌈ Method `BinFormat (uint16)` shall be defined as described in table 8.23. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin16 BinFormat(uint16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin16 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format.	

Table 8.23: Method definition - BinFormat (uint16)

8.2.14 BinFormat (int16)

[SWS_LOG_00033] conversion of an int16 into a binary value ⌈ Method `BinFormat (int16)` shall be defined as described in table 8.24. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin16 BinFormat(int16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin16 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

Table 8.24: Method definition - BinFormat (int16)

8.2.15 BinFormat (uint32)

[SWS_LOG_00034] conversion of a uint32 into a binary value ⌈ Method `BinFormat (uint32)` shall be defined as described in table 8.25. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin32 BinFormat(uint32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin32 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format.	

Table 8.25: Method definition - BinFormat (uint32)

8.2.16 BinFormat (int32)

[SWS_LOG_00035] conversion of an int32 into a binary value ⌈ Method `BinFormat (int32)` shall be defined as described in table 8.26. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin32 BinFormat(int32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin32 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

Table 8.26: Method definition - BinFormat (int32)

8.2.17 BinFormat (uint64)

[SWS_LOG_00036] conversion of a uint64 into a binary value ⌈ Method `BinFormat (uint64)` shall be defined as described in table 8.27. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin64 BinFormat(uint64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin64 type that has a built-in stream handler.	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format.	

Table 8.27: Method definition - BinFormat (uint64)

8.2.18 BinFormat (int64)

[SWS_LOG_00037] conversion of an int64 into a binary value ⌈ Method `BinFormat (int64)` shall be defined as described in table 8.28. ⌋*(RS_LT_00046)*

Service name:	BinFormat	
Syntax:	`LogBin64 BinFormat(int64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogBin64 type that has a built-in stream handler	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

Table 8.28: Method definition - BinFormat (int64)

8.2.19 RawBuffer

[SWS_LOG_00038] ⌈ Method `RawBuffer` shall be defined as described in table 8.29.
⌋(*RS_LT_00044*)

Service name:	RawBuffer	
Syntax:	`template <typename T> LogRawBuffer RawBuffer(const T& value) noexcept;`	
Parameters (in):	value	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	LogRawBuffer type that has a built-in stream handler	
Exceptions:	None	
Header file:	ara/log/logging.h	
Description:	Logs raw binary data by providing a buffer	

Table 8.29: Method definition - RawBuffer

8.3 Class definitions

8.3.1 Class LogStream

The Class LogStream represents a log message, allowing stream operators to be used for appending data.

Note:
Normally applications 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 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 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 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;
  std::string bar;
};

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

LogDebug () << MyCustomType{42, "The answer is"};
```

8.3.1.2 LogStream::Flush

[SWS_LOG_00039] ⌈ Method `LogStream::Flush` shall be defined as described in table 8.30. ⌋*(RS_LT_00049)*

Service name:	LogStream::Flush
Syntax:	`void Flush() noexcept`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Sends out the current log buffer and initiates a new message stream.

Table 8.30: Method definition - LogStream::Flush

Note:
Calling Flush() is only necessary if the `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), than flush operation will be anyway done internally by the destructor. It is important to note that the Flush() command does not empty the buffer, but it forwards its current contents to the logging framework.

8.3.1.3 Built-in operators for natively supported types:

[SWS_LOG_00040] ⌈ Operator `bool handler` shall be defined as described in table 8.31. ⌋*(RS_LT_00049)*

Service name:	bool handler
Syntax:	`LogStream& operator≪(bool value) noexcept`
Parameters (in):	bool value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.31: Operator definition - bool handler

[SWS_LOG_00041] ⌈ Operator `uint8_t handler` shall be defined as described in table 8.32. ⌋*(RS_LT_00049)*

Service name:	uint8_t handler
Syntax:	`LogStream& operator≪(uint8_t value) noexcept`
Parameters (in):	uint8_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`

Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.32: Operator definition - uint8_t handler

[SWS_LOG_00042] ⌈ Operator `uint16_t handler` shall be defined as described in table 8.33. ⌋*(RS_LT_00049)*

Service name:	uint16_t handler
Syntax:	`LogStream& operator<<(uint16_t value) noexcept`
Parameters (in):	uint16_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.33: Operator definition - uint16_t handler

[SWS_LOG_00043] ⌈ Operator `uint32_t handler` shall be defined as described in table 8.34. ⌋*(RS_LT_00049)*

Service name:	uint32_t handler
Syntax:	`LogStream& operator<<(uint32_t value) noexcept`
Parameters (in):	uint32_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.34: Operator definition - uint32_t handler

[SWS_LOG_00044] ⌈ Operator `uint64_t handler` shall be defined as described in table 8.35. ⌋*(RS_LT_00049)*

Service name:	uint64_t handler
Syntax:	`LogStream& operator<<(uint64_t value) noexcept`
Parameters (in):	uint64_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.35: Operator definition - uint64_t handler

[SWS_LOG_00045] ⌈ Operator `int8_t handler` shall be defined as described in table 8.36. ⌋*(RS_LT_00049)*

Service name:	int8_t handler
Syntax:	`LogStream& operator≪(int8_t value) noexcept`
Parameters (in):	int8_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.36: Operator definition - int8_t handler

[SWS_LOG_00046] ⌈ Operator `int16_t handler` shall be defined as described in table 8.37. ⌋*(RS_LT_00049)*

Service name:	int16_t handler
Syntax:	`LogStream& operator≪(int16_t value) noexcept`
Parameters (in):	int16_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.37: Operator definition - int16_t handler

[SWS_LOG_00047] ⌈ Operator `int32_t handler` shall be defined as described in table 8.38. ⌋*(RS_LT_00049)*

Service name:	int32_t handler
Syntax:	`LogStream& operator≪(int32_t value) noexcept`
Parameters (in):	int32_t value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.38: Operator definition - int32_t handler

[SWS_LOG_00048] ⌈ Operator `int64_t handler` shall be defined as described in table 8.39. ⌋*(RS_LT_00049)*

Service name:	int64_t handler
Syntax:	`LogStream& operator≪(int64_t value) noexcept`
Parameters (in):	int64_t value
Parameters (inout):	None
Parameters (out):	None

Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.39: Operator definition - int64_t handler

[SWS_LOG_00049] ⌈ Operator `float handler` shall be defined as described in table 8.40. ⌋*(RS_LT_00049)*

Service name:	float handler
Syntax:	`LogStream& operator<<(float value) noexcept`
Parameters (in):	float value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.40: Operator definition - float handler

[SWS_LOG_00050] ⌈ Operator `double handler` shall be defined as described in table 8.41. ⌋*(RS_LT_00049)*

Service name:	double handler
Syntax:	`LogStream& operator<<(double value) noexcept`
Parameters (in):	double value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.41: Operator definition - double handler

[SWS_LOG_00051] ⌈ Operator `null-terminated char string handler` shall be defined as described in table 8.42. ⌋*(RS_LT_00049)*

Service name:	null-terminated char string handler
Syntax:	`LogStream& operator<<(const char* const value) noexcept`
Parameters (in):	char* value
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.42: Operator definition - null-terminated char string handler

8.3.1.4 Built-in operators for conversion types:

[SWS_LOG_00053] ⌈ Operator `LogHex8 handler` shall be defined as described in table 8.43. ⌋*(RS_LT_00046)*

Service name:	LogHex handler
Syntax:	`LogStream& operator<<(const LogHex8& value) noexcept`
Parameters (in):	Reference to a `LogHex8 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.43: Operator definition - LogHex handler

[SWS_LOG_00054] ⌈ Operator `LogHex16 handler` shall be defined as described in table 8.44. ⌋*(RS_LT_00046)*

Service name:	LogHex16 handler
Syntax:	`LogStream& operator<<(const LogHex16& value) noexcept`
Parameters (in):	Reference to a `LogHex16 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.44: Operator definition - LogHex16 handler

[SWS_LOG_00055] ⌈ Operator `LogHex32 handler` shall be defined as described in table 8.45. ⌋*(RS_LT_00046)*

Service name:	LogHex32 handler
Syntax:	`LogStream& operator<<(const LogHex32& value) noexcept`
Parameters (in):	Reference to a `LogHex32 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.45: Operator definition - LogHex32 handler

[SWS_LOG_00056] ⌈ Operator `LogHex64 handler` shall be defined as described in table 8.46. ⌋*(RS_LT_00046)*

Service name:	LogHex64 handler
Syntax:	`LogStream& operator<<(const LogHex64& value) noexcept`
Parameters (in):	Reference to a `LogHex64 value`

Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.46: Operator definition - LogHex64 handler

[SWS_LOG_00057] ⌈ Operator `LogBin8 handler` shall be defined as described in table 8.47. ⌋*(RS_LT_00046)*

Service name:	LogBin8 handler
Syntax:	`LogStream& operator≪(const LogBin8& value) noexcept`
Parameters (in):	Reference to a `LogBin8 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.47: Operator definition - LogBin8 handler

[SWS_LOG_00058] ⌈ Operator `LogBin16 handler` shall be defined as described in table 8.48. ⌋*(RS_LT_00046)*

Service name:	LogBin16 handler
Syntax:	`LogStream& operator≪(const LogBin16& value) noexcept`
Parameters (in):	Reference to a `LogBin16 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.48: Operator definition - LogBin16 handler

[SWS_LOG_00059] ⌈ Operator `LogBin32 handler` shall be defined as described in table 8.49. ⌋*(RS_LT_00046)*

Service name:	LogBin32 handler
Syntax:	`LogStream& operator≪(const LogBin32& value) noexcept`
Parameters (in):	Reference to a `LogBin32 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.49: Operator definition - LogBin32 handler

[SWS_LOG_00060] ⌈ Operator `LogBin64 handler` shall be defined as described in table 8.50. ⌋*(RS_LT_00046)*

Service name:	LogBin64 handler
Syntax:	`LogStream& operator≪(const LogBin64& value) noexcept`
Parameters (in):	Reference to a `LogBin64 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends given value to the internal message buffer.

Table 8.50: Operator definition - LogBin64 handler

8.3.1.5 Built-in operators for extra types:

[SWS_LOG_00061] ⌈ Operator `LogRawBuffer handler` shall be defined as described in table 8.51. ⌋*(RS_LT_00044)*

Service name:	LogRawBuffer handler
Syntax:	`LogStream& operator≪(const LogRawBuffer& value) noexcept`
Parameters (in):	Reference to a `LogRawBuffer value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends plain binary data into message buffer.

Table 8.51: Operator definition - LogRawBuffer handler

[SWS_LOG_00062] ⌈ Operator `std::string handler` shall be defined as described in table 8.52. ⌋*(RS_LT_00049)*

Service name:	std::string handler
Syntax:	`LogStream& operator≪(const std::string& value) noexcept`
Parameters (in):	Reference to a `std::string value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends STL string to message buffer.

Table 8.52: Operator definition - std::string handler

[SWS_LOG_00063] ⌈ Operator `LogLevel handler` shall be defined as described in table 8.52. ⌋*(RS_LT_00049)*

Service name:	LogLevel handler
Syntax:	`LogStream& operator<<(LogLevel value) noexcept`
Parameters (in):	Reference to a `LogLevel value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Header file:	ara/log/logstream.h
Description:	Appends LogLevel enum parameter as text into message.

Table 8.53: Operator definition - LogLevel handler

8.3.2 Class Logger

The Class Logger represents a DLT logger context. DLT defines contexts which can be seen as logger instances within one application or process scope.

A context will be automatically registered against the DLT 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 applications are not supposed to call the Logger constructor themselves.

8.3.2.1 Logger::LogFatal

[SWS_LOG_00064] ⌈ LogStream `LogFatal` shall be defined as described in table 8.54. ⌋*(RS_LT_00049)*

Service name:	Logger::LogFatal
Syntax:	`LogStream LogFatal () noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Fatal
Exceptions:	None
Header file:	ara/log/logger.h
Description:	Creates a LogStream object of Fatal severity that has to be used by passing arguments via the input stream operator "≪" .

Table 8.54: LogStream definition - LogFatal

8.3.2.2 Logger::LogError

[SWS_LOG_00065] ⌈ LogStream `LogError` shall be defined as described in table 8.55. ⌋*(RS_LT_00049)*

Service name:	Logger::LogError
Syntax:	`LogStream LogError () noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Error
Exceptions:	None
Header file:	ara/log/logger.h
Description:	Creates a LogStream object of Error severity that has to be used by passing arguments via the input stream operator "≪".

Table 8.55: LogStream definition - LogError

8.3.2.3 Logger::LogWarn

[SWS_LOG_00066] ⌈ LogStream `LogWarn` shall be defined as described in table 8.56. ⌋*(RS_LT_00049)*

Service name:	Logger::LogWarn
Syntax:	`LogStream LogWarn() noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Warn
Exceptions:	None
Header file:	ara/log/logger.h
Description:	Creates a LogStream object of Warn severity that has to be used by passing arguments via the input stream operator "$\ll$" .

Table 8.56: LogStream definition - LogWarn

8.3.2.4 Logger::LogInfo

[SWS_LOG_00067] ⌈ LogStream `LogInfo` shall be defined as described in table 8.57. ⌋*(RS_LT_00049)*

Service name:	Logger::LogInfo
Syntax:	`LogStream LogInfo() noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Info
Exceptions:	None
Header file:	ara/log/logger.h
Description:	Creates a LogStream object of Info severity that has to be used by passing arguments via the input stream operator "$\ll$" .

Table 8.57: LogStream definition - LogInfo

8.3.2.5 Logger::LogDebug

[SWS_LOG_00068] ⌈ LogStream `LogDebug` shall be defined as described in table 8.58. ⌋*(RS_LT_00049)*

Service name:	Logger::LogDebug
Syntax:	`LogStream LogDebug() noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Debug
Exceptions:	None
Header file:	ara/log/logger.h

Description:	Creates a LogStream object of Debug severity that has to be used by passing arguments via the input stream operator "≪" .

Table 8.58: LogStream definition - LogDebug

8.3.2.6 Logger::LogVerbose

[SWS_LOG_00069] ⌈ LogStream `LogVerbose` shall be defined as described in table 8.59. ⌋*(RS_LT_00049)*

Service name:	Logger::LogVerbose
Syntax:	`LogStream LogVerbose() noexcept;`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	LogStream object of log level Verbose
Exceptions:	None
Header file:	ara/log/logger.h
Description:	Creates a LogStream object of Verbose severity that has to be used by passing arguments via the input stream operator "≪" .

Table 8.59: LogStream definition - LogVerbose

8.3.2.7 Logger::IsLogEnabled

[SWS_LOG_00070] ⌈ Method `Logger::IsLogEnabled` shall be defined as described in table 8.60. ⌋*(RS_LT_00045)*

Service name:	Logger::IsLogEnabled
Syntax:	`bool IsLogEnabled(LogLevel logLevel) const noexcept;`
Parameters (in):	logLevel
Parameters (inout):	None
Parameters (out):	None
Return value:	True if desired log level satisfies the configured reporting level, otherwise False.
Exceptions:	None
Header file:	ara/log/logger.h
Description:	The Application can check if the current configured log will pass desired log level..

Table 8.60: Method definition - Logger::IsLogEnabled

9 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers, as well as Constraints (if it applies) and Published Information.

The Log and Trace configuration can be found in chapter 7.5 of AUTOSAR Manifest Specification [4].