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	R19-11

Document Change History

Date	Release	Changed by	Description
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

Table of Contents

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, 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 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].

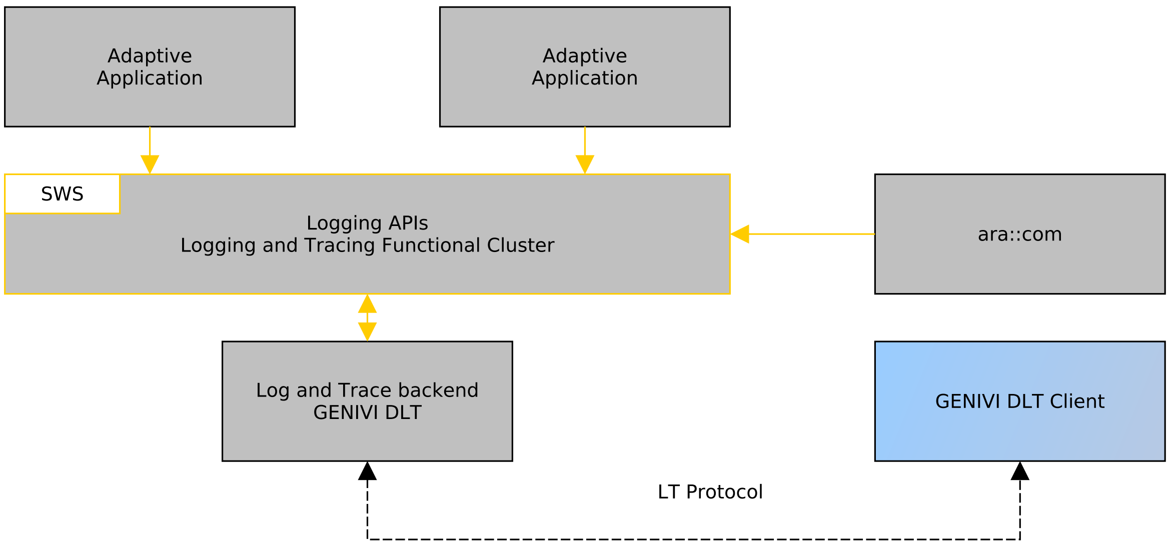

Figure 1.1: Architecture overview

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

2 Acronyms and 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

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

Table 2.1: Glossary-defined Technical Terms

3 Related documentation

3.1 Input documents

[1] Log and Trace Protocol Specification
AUTOSAR_PRS_LogAndTraceProtocol

[2] Glossary
AUTOSAR_TR_Glossary

[3] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[4] Requirements on Log and Trace
AUTOSAR_RS_LogAndTrace

[5] Specification of Time Synchronization for Adaptive Platform
AUTOSAR_SWS_TimeSync

4 Constraints and assumptions

4.1 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

There are no dependencies to other Functional Clusters.

5.1 Platform dependencies

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

6 Requirements Tracing

The following table references the requirements specified in RS Log And Trace [4] 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_00017]	Each log and trace message shall contain a timestamp, which will be added to the message during reception of the message in the LT module.	[SWS_LOG_00082] [SWS_LOG_00083] [SWS_LOG_00091]
[RS_LT_00030]	Monitoring and shaping of LT log and trace event amount .	[SWS_LOG_00095]
[RS_LT_00044]	Provide raw buffer content.	[SWS_LOG_00014] [SWS_LOG_00038]
[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_00051] [SWS_LOG_00053] [SWS_LOG_00054] [SWS_LOG_00055] [SWS_LOG_00056] [SWS_LOG_00057] [SWS_LOG_00058] [SWS_LOG_00059] [SWS_LOG_00060] [SWS_LOG_00061] [SWS_LOG_00062] [SWS_LOG_00063] [SWS_LOG_00108] [SWS_LOG_00109] [SWS_LOG_00110] [SWS_LOG_00111] [SWS_LOG_00112] [SWS_LOG_00113] [SWS_LOG_00114] [SWS_LOG_00115] [SWS_LOG_00116] [SWS_LOG_00120] [SWS_LOG_00124]
[RS_LT_00047]	Initialization and registration.	[SWS_LOG_00003] [SWS_LOG_00004]
[RS_LT_00048]	Meta information about Applications.	[SWS_LOG_00004]

Requirement	Description	Satisfied by
[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_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] [SWS_LOG_00098] [SWS_LOG_00101]
[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++ 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 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) Log and trace network bandwith limitation (see chapter 7.5)

7.1 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 amount of contexts, but at least one – the default context.

Machine-specific configuration settings for the Log and Trace functional cluster are collected in LogAndTraceInstantiation. The Application processes 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 predifiend 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 predifiend value is set

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, 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 logTraceProcessId in the Process 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.

7.1.2 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 logTraceProcessDesc in the Process is an optional setting that allows to describe the logTraceProcessId as descriptive text.

7.1.3 Default Log Level

The Log severity level represents the severity of the log messages. Severity levels are defined in chapter 7.2. logTraceDefaultLogLevel in the Process 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 messages is 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, a file on the file system or the communication bus. The system integrator is responsbile to populate this information in the application execution manifest. A direct API for dynamically changing this value for development purposes is provided. In the AUTOSAR Meta-Model the logTraceLogMode is equivalent to the log mode described here, for more information see [3]. logTraceLogMode in the Process defines the destination to which the log messages will be forwarded.

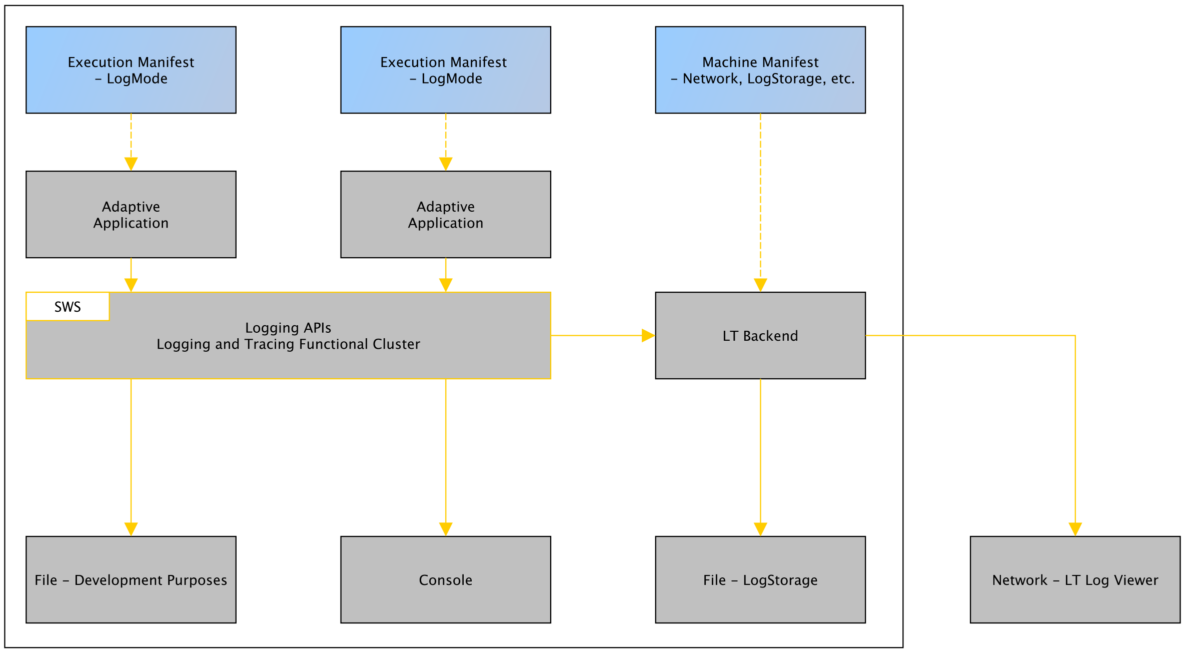

Figure 7.1: Log mode

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 `EthernetNetworkConfiguration` that is aggregated by the `LogAndTraceInstantiation` in the role `networkConfiguration`.

7.1.4.1 Log File Path

In case the log mode is set to log to a file, a destination directory path needs to be provided. `logTraceFilePath` in the `Process` 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.

7.1.5 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 process`es 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 process`es 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`.

7.1.6 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.

7.1.7 Initialization of the Logging framework

Before the logging information can be processed, the `Logging framework` needs to be initialized. The logging functionality is initialized by `ara::core::Initialize()` when the parameter `Executable.loggingBehavior` is set. In order to initialize the `Logging framework`, the mandatory information needs to be provided to the `Logging framework`. The essential information for the `Logging framework` is extracted from the application execution manifest and the AUTOSAR Meta-Model as described above.

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_00001]{DRAFT} ⌈All messages logged before the initialization of the `Logging framework` is done shall be stored inside a buffer with a limited size, i.e the oldest entries are discarded if the buffer is exceeded. The size of the buffer is an implementation detail.⌋*(RS_LT_00052)*

[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_00003]{DRAFT} ⌈Before `Log message`s can be processed, the `CreateLogger()` function needs to be called. The first call of the function initializes the `Logging framework` for the application and creates a valid logging context.⌋*(RS_LT_00047)*

[SWS_LOG_00004]{DRAFT} ⌈The application execution manifest should provide the following information for the `Logging framework` to be initialized:

- A unique application ID
- An application description
- The default `Log severity 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]{DRAFT} ⌈The function `CreateLogger()` shall create a logger context 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 logger context shall be available.⌋*(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]{DRAFT} ⌈By calling `CreateLogger()`, the following parameters need to be provided:

- The context ID
- The context description
- The `Log severity level` (as an optional parameter, defaults to `LogLevel::kWarn`)

⌋*(RS_LT_00050)*

[SWS_LOG_00007]{DRAFT} ⌈`Application process`es should be able to check if a desired `Log severity 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 `Logging framework` offers stream based API for `Log 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 `Log severity level`, a separate function call is foreseen.

The following `Log severity level`s 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 information used during development - call stacks, line numbers or raw data to perform stepwise problem localization)
- Verbose (Verbose information with insight into the behavior of the system without exposing any critical or sensitive data)

Note:
Off is not applicable for `Log message`s. This level can be used to set reporting level for the `Logging framework` either initially through the configuration of the application or during runtime.

Design Rationale:
For having separate functions per `Log severity level` vs. passing the 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 messages` 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 `Logging 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 `Logging 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.

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;
```

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

New line: Because of convenience purposes the Logging framework automatically appends a newline to the Log message.

Multiple payload arguments: When one message consists of more than one payload argument, payload arguments a separated by single whitespaces for console output.

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

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

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

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

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

[SWS_LOG_00013]{DRAFT} ⌈To initiate a Log message with the Log level `Verbose`, the API `LogVerbose()` shall be called. This API returns a `LogStream` object that has to be used by passing arguments via the insert stream operator "<<".⌋*(RS_-LT_00049)*

[SWS_LOG_00014]{DRAFT} ⌈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_00120]{DRAFT} ⌈Dedicated conversion functions are provided for conversion of positive decimal numbers into a string with hexadecimal or binary representation.⌋*(RS_LT_00046)*

[SWS_LOG_00015]{DRAFT} ⌈Dedicated conversion functions are provided for conversion of decimal numbers into a string with hexadecimal or binary representation, where the most significant bit shall be set to '1' for negative numbers.⌋*(RS_LT_00046)*

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

[SWS_LOG_00017]{DRAFT} ⌈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 `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 [4] for more information).

The synchronized time base is supplied by the Time Synchronization `Functional Cluster`. The `now()` method is used by the `Adaptive Application`s in order to retrieve the current time from the TS (refer to [5] for more information).

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

[SWS_LOG_00082]{DRAFT} ⌈`Log and Trace` shall have accesss to a synchronized time base. The attribute `timeBaseResource` in `LogAndTraceInstantiation` shall be used to identify the time base.⌋*(RS_LT_00017)*

[SWS_LOG_00083]{DRAFT} ⌈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.⌋*(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_00017)*

7.5 Log and Trace Network Bandwith Limitation

[SWS_LOG_00095]{DRAFT} ⌈When `Log and Trace` receives a high load of trace information, generated at the same time, from multiple `Adaptive Application`s, it shall buffer this data internally so it can be sent continuously and so that no information is lost.⌋*(RS_LT_00030)*

7.6 Functional Cluster Lifecyle

7.6.1 Shutdown

[SWS_LOG_00122]{DRAFT} ⌈When ara::core::Deinitialize() is called, the `Logging framework` shall make sure, that no new client connections can be established.⌋*()*

[SWS_LOG_00123]{DRAFT} ⌈When ara::core::Deinitialize() is called, the `Logging framework` shall take care that all remaining messages in the buffer can be collected, if a client is connected.⌋*()*

8 API specification

8.1 Type definitions

8.1.1 LogLevel

[SWS_LOG_00018]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::log::LogLevel	
Scope:	namespace ara::log	
Underlying type:	uint8_t	
Syntax:	`enum class LogLevel : 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)
Header file:	#include "ara/log/common.h"	
Description:	List of possible severity levels .	

⌋*(RS_LT_00049)*

8.1.2 LogMode

[SWS_LOG_00019]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::log::LogMode	
Scope:	namespace ara::log	
Underlying type:	uint8_t	
Syntax:	`enum class LogMode : uint8_t {...};`	
Values:	kRemote= 0x01	Sent remotely.
	kFile= 0x02	Save to file.
	kConsole= 0x04	Forward to console.
Header file:	#include "ara/log/common.h"	
Description:	Log mode. Flags, used to configure the sink for log messages.	
Notes:	In order to combine flags, at least the OR and AND operators needs to be provided for this type.	

⌋*(RS_LT_00051)*

8.1.3 LogHex8

[SWS_LOG_00108]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogHex8
Scope:	namespace ara::log
Syntax:	`struct LogHex8 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 8 bit hexadecimal value data type . Helper struct that is utilized as custom type. Holds an integer value that will be logged with a special format.

⌋*(RS_LT_00046)*

8.1.4 LogHex16

[SWS_LOG_00109]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogHex16
Scope:	namespace ara::log
Syntax:	`struct LogHex16 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 16 bit hexadecimal value data type .

⌋*(RS_LT_00046)*

8.1.5 LogHex32

[SWS_LOG_00110]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogHex32
Scope:	namespace ara::log
Syntax:	`struct LogHex32 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 32 bit hexadecimal value data type .

⌋*(RS_LT_00046)*

8.1.6 LogHex64

[SWS_LOG_00111]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogHex64
Scope:	namespace ara::log
Syntax:	`struct LogHex64 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 64 bit hexadecimal value data type .

⌋*(RS_LT_00046)*

8.1.7 LogBin8

[SWS_LOG_00112]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogBin8
Scope:	namespace ara::log
Syntax:	`struct LogBin8 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 8 bit binary data type .

⌋*(RS_LT_00046)*

8.1.8 LogBin16

[SWS_LOG_00113]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogBin16
Scope:	namespace ara::log
Syntax:	`struct LogBin16 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 16 bit binary data type .

⌋*(RS_LT_00046)*

8.1.9 LogBin32

[SWS_LOG_00114]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogBin32
Scope:	namespace ara::log
Syntax:	`struct LogBin32 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 32 bit binary data type .

⌋*(RS_LT_00046)*

8.1.10 LogBin64

[SWS_LOG_00115]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogBin64
Scope:	namespace ara::log
Syntax:	`struct LogBin64 {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a 64 bit binary data type .

⌋*(RS_LT_00046)*

8.1.11 LogRawBuffer

[SWS_LOG_00116]{DRAFT} ⌈

Kind:	struct
Symbol:	ara::log::LogRawBuffer
Scope:	namespace ara::log
Syntax:	`struct LogRawBuffer {...};`
Header file:	#include "ara/log/logstream.h"
Description:	Represents a raw data buffer of a limited size. Helper struct that is utilized as custom type. Holds a pointer to some data and the size which is to be logged as raw data.

⌋*(RS_LT_00046)*

8.1.12 ClientState

[SWS_LOG_00098]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::log::ClientState	
Scope:	namespace ara::log	
Underlying type:	int8_t	
Syntax:	`enum class ClientState :  int8_t {...};`	
Values:	kUnknown= -1	–
	kNotConnected	–
	kConnected	–
Header file:	#include "ara/log/common.h"	
Description:	Client state representing the connection state of an external client. .	

⌋*(RS_LT_00050)*

8.2 Function definitions

8.2.1 CreateLogger

[SWS_LOG_00021]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::CreateLogger(ara::core::StringView ctxId, ara::core::StringView ctxDescription, LogLevel ctxDefLogLevel=LogLevel::kWarn)	
Scope:	namespace ara::log	
Syntax:	`Logger& CreateLogger (ara::core::StringView ctxId, ara::core::StringView ctxDescription, LogLevel ctxDefLogLevel=LogLevel::kWarn) 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	
Header file:	#include "ara/log/logging.h"	
Description:	Creates a Logger object, holding the context which is registered in the Logging framework.	

⌋*(RS_LT_00050)*

8.2.2 HexFormat (uint8)

[SWS_LOG_00022]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(uint8_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex8 HexFormat (uint8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex8	LogHex8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint8 into a hexadecimal value. Negatives are represented in 2's complement. The number of represented digits depends on the overloaded parameter type length.	
Notes:	Logs decimal numbers in hexadecimal format.	

⌋*(RS_LT_00046)*

8.2.3 HexFormat (int8)

[SWS_LOG_00023]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(int8_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex8 HexFormat (int8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex8	LogHex8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int8 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.4 HexFormat (uint16)

[SWS_LOG_00024]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(uint16_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex16 HexFormat (uint16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex16	LogHex16 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint16 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format.	

⌋*(RS_LT_00046)*

8.2.5 HexFormat (int16)

[SWS_LOG_00025]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(int16_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex16 HexFormat (int16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex16	LogHex16 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int16 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.6 HexFormat (uint32)

[SWS_LOG_00026]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(uint32_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex32 HexFormat (uint32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex32	LogHex32 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint32 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format.	

⌋*(RS_LT_00046)*

8.2.7 HexFormat (int32)

[SWS_LOG_00027]{DRAFT} ⌈

Kind:	function
Symbol:	ara::log::HexFormat(int32_t value)
Scope:	namespace ara::log

▽

△

Syntax:	`constexpr LogHex32 HexFormat (int32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex32	LogHex32 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int32 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.8 HexFormat (uint64)

[SWS_LOG_00028]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(uint64_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex64 HexFormat (uint64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.
Return value:	LogHex64	LogHex64 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint64 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format.	

⌋*(RS_LT_00046)*

8.2.9 HexFormat (int64)

[SWS_LOG_00029]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::HexFormat(int64_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogHex64 HexFormat (int64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into hexadecimal number system.

▽

△

Return value:	LogHex64	LogHex64 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int64 into a hexadecimal value.	
Notes:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.10 BinFormat (uint8)

[SWS_LOG_00030]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(uint8_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin8 BinFormat (uint8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin8	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint8 into a binary value.	
Notes:	Logs decimal numbers in binary format.	

⌋*(RS_LT_00046)*

8.2.11 BinFormat (int8)

[SWS_LOG_00031]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(int8_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin8 BinFormat (int8_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin8	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	

▽

△

Header file:	#include "ara/log/logging.h"
Description:	Conversion of a int8 into a binary value.
Notes:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.

⌋*(RS_LT_00046)*

8.2.12 BinFormat (uint16)

[SWS_LOG_00032]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(uint16_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin16 BinFormat (uint16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin16	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint16 into a binary value.	
Notes:	Logs decimal numbers in binary format.	

⌋*(RS_LT_00046)*

8.2.13 BinFormat (int16)

[SWS_LOG_00033]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(int16_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin16 BinFormat (int16_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin16	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int16 into a binary value.	
Notes:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.14 BinFormat (uint32)

[SWS_LOG_00034]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(uint32_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin32 BinFormat (uint32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin32	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint32 into a binary value.	
Notes:	Logs decimal numbers in binary format.	

⌋*(RS_LT_00046)*

8.2.15 BinFormat (int32)

[SWS_LOG_00035]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(int32_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin32 BinFormat (int32_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin32	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int32 into a binary value.	
Notes:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.16 BinFormat (uint64)

[SWS_LOG_00036]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(uint64_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin64 BinFormat (uint64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin64	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a uint64 into a binary value.	
Notes:	Logs decimal numbers in binary format.	

⌋*(RS_LT_00046)*

8.2.17 BinFormat (int64)

[SWS_LOG_00037]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::BinFormat(int64_t value)	
Scope:	namespace ara::log	
Syntax:	`constexpr LogBin64 BinFormat (int64_t value) noexcept;`	
Parameters (in):	value	Decimal number to be converted into a binary value.
Return value:	LogBin64	LogBin8 type that has a built-in stream handler.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Conversion of a int64 into a binary value.	
Notes:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋*(RS_LT_00046)*

8.2.18 RawBuffer

[SWS_LOG_00038]{DRAFT} ⌈

Kind:	function
Symbol:	ara::log::RawBuffer(const T &value)
Scope:	namespace ara::log

▽

△

Syntax:	`template <typename T, typename std::enable_if<!std::is_pointer< T >::value, std::nullptr_t >::type = nullptr> constexpr LogRawBuffer RawBuffer (const T &value) noexcept;`	
Template param:	T	The type of the contents of value.
Parameters (in):	value	the value to convert to raw data.
Return value:	LogRawBuffer	LogRawBuffer type that has a built-in stream handler.
Exception Safety:	noexcept	
Header file:	#include "ara/log/logging.h"	
Description:	Logs raw binary data by providing a buffer.	
Notes:	T can take an arbitrary type, though it is not possible to specify a pointer as an argument. The maximum size of the provided data that can be processed depends on the underlying back-end implementation.	

⌋*(RS_LT_00044)*

8.2.19 remoteClientState

[SWS_LOG_00101]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::remoteClientState()	
Scope:	namespace ara::log	
Syntax:	`ClientState remoteClientState () const noexcept;`	
Return value:	ClientState	The current client state.
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logging.h"	
Description:	Fetches the connection state from the DLT back-end of a possibly available remote client.	

⌋*(RS_LT_00050)*

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

8.3.1.2 LogStream::Flush

[SWS_LOG_00039]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

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) then the flushing operation will be done internally by the destructor. It is important to note that the `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} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00041]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00042]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00043]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00044]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00045]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00046]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00047]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00048]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00049]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00050]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

[SWS_LOG_00061]{DRAFT} ⌈

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

⌋*(RS_LT_00046)*

8.3.1.4 Built-in operators for conversion types

[SWS_LOG_00053]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogHex8 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogHex8 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	

▽

△

Header file:	#include "ara/log/logstream.h"
Description:	Writes unsigned int parameter into message, formatted as hexadecimal 8 digits.

⌋*(RS_LT_00046)*

[SWS_LOG_00054]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogHex16 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogHex16 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as hexadecimal 16 digits.	

⌋*(RS_LT_00046)*

[SWS_LOG_00055]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogHex32 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogHex32 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as hexadecimal 32 digits.	

⌋*(RS_LT_00046)*

[SWS_LOG_00056]{DRAFT} ⌈

Kind:	function
Symbol:	ara::log::LogStream::operator<<(const LogHex64 &value)
Scope:	class ara::log::LogStream
Syntax:	`LogStream& operator<< (const LogHex64 &value) noexcept;`

▽

△

Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as hexadecimal 64 digits.	

⌋(*RS_LT_00046*)

[SWS_LOG_00057]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogBin8 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogBin8 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as binary 8 digits.	

⌋(*RS_LT_00046*)

[SWS_LOG_00058]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogBin16 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogBin16 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as binary 16 digits.	

⌋(*RS_LT_00046*)

[SWS_LOG_00059]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogBin32 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogBin32 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as binary 32 digits.	

⌋(*RS_LT_00046*)

[SWS_LOG_00060]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const LogBin64 &value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const LogBin64 &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes unsigned int parameter into message, formatted as binary 64 digits.	

⌋(*RS_LT_00046*)

8.3.1.5 Built-in operators for extra types

[SWS_LOG_00062]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const ara::core::StringView value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const ara::core::StringView value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	

▽

△

Header file:	#include "ara/log/logstream.h"
Description:	Writes ara::core::StringView into message.

⌋(*RS_LT_00046*)

[SWS_LOG_00051]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::LogStream::operator<<(const char *const value)	
Scope:	class ara::log::LogStream	
Syntax:	`LogStream& operator<< (const char *const value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes null terminated UTF8 string into message. (NOT sPECIFIED. WILL BE REMOVED IN FUTURE!)	

⌋(*RS_LT_00046*)

[SWS_LOG_00063]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::operator<<(LogStream &out, LogLevel value)	
Scope:	namespace ara::log	
Syntax:	`LogStream& operator<< (LogStream &out, LogLevel value) noexcept;`	
Parameters (in):	value	LogLevel enum parameter as text to be appended to the internal message buffer.
DIRECTION NOT DEFINED	out	–
Return value:	LogStream &	–
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Appends LogLevel enum parameter as text into message.	

⌋(*RS_LT_00046*)

[SWS_LOG_00124]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::operator<<(const ara::core::ErrorCode &value)	
Scope:	namespace ara::log	
Syntax:	`LogStream& operator<< (const ara::core::ErrorCode &value) noexcept;`	
Parameters (in):	value	Value to be appended to the internal message buffer.
Return value:	LogStream &	*this
Exception Safety:	noexcept	
Thread Safety:	reentrant	
Header file:	#include "ara/log/logstream.h"	
Description:	Writes an ara::core::ErrorCode into the message, containing a String holding the results of ErrorCode:Domain().Name() (i.e. the ErrorDomain's Shortname), and the integral error code number.	

⌋*(RS_LT_00046)*

8.3.2 Class Logger

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

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} ⌈

Kind:	function	
Symbol:	ara::log::Logger::LogFatal()	
Scope:	class ara::log::Logger	
Syntax:	`LogStream LogFatal () noexcept;`	
Return value:	LogStream	LogStream object of Fatal severity.
Exception Safety:	noexcept	
Header file:	#include "ara/log/logger.h"	
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_00049)*

8.3.2.2 Logger::LogError

[SWS_LOG_00065]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

8.3.2.3 Logger::LogWarn

[SWS_LOG_00066]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

8.3.2.4 Logger::LogInfo

[SWS_LOG_00067]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

8.3.2.5 Logger::LogDebug

[SWS_LOG_00068]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

8.3.2.6 Logger::LogVerbose

[SWS_LOG_00069]{DRAFT} ⌈

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

⌋*(RS_LT_00049)*

8.3.2.7 Logger::IsEnabled

[SWS_LOG_00070]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::log::Logger::IsEnabled(LogLevel logLevel)	
Scope:	class ara::log::Logger	
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
Header file:	#include "ara/log/logger.h"
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_00045)*

A Mentioned Class Tables

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

Class	**EthernetNetworkConfiguration**			
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.ManifestKind=MachineManifest
	atp.Status=draft			

Base	*ARObject*, *NetworkConfiguration*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
communication Connector	EthernetCommunication Connector	0..1	ref	Reference to the CommunicationConnector (VLAN) for which the network configuration is defined. **Tags:**atp.Status=draft
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.
tcpPort	PositiveInteger	0..1	attr	This attribute allows to configure a tcp port number.
udpPort	PositiveInteger	0..1	attr	This attribute allows to configure a udp port number.

Table A.1: EthernetNetworkConfiguration

Class	**LogAndTraceInstantiation**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::PlatformModuleDeployment::AdaptiveModule Implementation			

Note	This meta-class defines the attributes for the Log&Trace configuration on a specific machine.
	
	Tags:
	atp.ManifestKind=MachineManifest
	atp.Status=draft			

Base	*ARObject*, *AdaptiveModuleInstantiation*, *Identifiable*, *MultilanguageReferrable*, *NonOsModule Instantiation*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
network Configuration	NetworkConfiguration	*	aggr	Network configuration for transmission of log & trace messages. **Tags:**atp.Status=draft
timeBase Resource	TimeBaseResource	0..1	ref	This reference is used to describe to which time base the 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. **Tags:**atp.Status=draft

Table A.2: LogAndTraceInstantiation

Class	**Machine**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::MachineManifest			
Note	Machine that represents an Adaptive Autosar Software Stack. **Tags:** atp.ManifestKind=MachineManifest atp.Status=draft atp.recommendedPackage=Machines			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
default Application Timeout	EnterExitTimeout	0..1	aggr	This aggration defines a default timeout in the context of a given Machine with respect to the launching and termination of applications. **Tags:**atp.Status=draft
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 atp.Status=draft
functionGroup	ModeDeclarationGroup Prototype	*	aggr	This aggregation represents the collection of function groups of the enclosing Machine. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel atp.Status=draft vh.latestBindingTime=preCompileTime
hwElement	HwElement	*	ref	This reference is used to describe the hardware resources of the machine. **Stereotypes:** atpUriDef **Tags:**atp.Status=draft
machineDesign	MachineDesign	1	ref	Reference to the MachineDesign this Machine is implementing. **Tags:**atp.Status=draft
module Instantiation	AdaptiveModule Instantiation	*	aggr	Configuration of Adaptive Autosar module instances that are running on the machine. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=shortName atp.Status=draft
processor	Processor	1..*	aggr	This represents the collection of processors owned by the enclosing machine. **Tags:**atp.Status=draft
secure Communication Deployment	SecureCommunication Deployment	*	aggr	Deployment of secure communication protocol configuration settings to crypto module entities. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=shortName atp.Status=draft

▽

△

Class	**Machine**			
trustedPlatform Executable LaunchBehavior	TrustedPlatform ExecutableLaunch BehaviorEnum	1	attr	This attribute controls the behavior of how authentication affects the ability to launch for each Executable.

Table A.3: Machine

Class	**Process**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ExecutionManifest			

Note	This meta-class provides information required to execute the referenced executable.
	Tags:
	atp.ManifestKind=ExecutionManifest
	atp.Status=draft
	atp.recommendedPackage=Processes			

Base	*ARElement*, *ARObject*, *AbstractExecutionContext*, *AtpClassifier*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadablePackageElement*			
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. **Tags:**atp.Status=draft
deterministic Client	DeterministicClient	0..1	ref	This reference adds further execution characteristics for deterministic clients. **Tags:**atp.Status=draft
executable	Executable	0..1	ref	Reference to executable that is executed in the process. **Stereotypes:** atpUriDef **Tags:**atp.Status=draft
logTraceDefault LogLevel	LogTraceDefaultLog LevelEnum	0..1	attr	This attribute allows to set the initial log reporting level for a logTraceProcessId (ApplicationId).
logTraceFile Path	UriString	0..1	attr	This attribute defines the destination file to which the logging information is passed.
logTraceLog Mode	LogTraceLogMode Enum	*	attr	This attribute defines the destination of log messages provided by the process.
logTrace ProcessDesc	String	0..1	attr	This attribute can be used to describe the logTrace ProcessId that is used in the log and trace message in more detail.
logTrace ProcessId	String	0..1	attr	This attribute identifies the process in the log and trace message (ApplicationId).
numberOf RestartAttempts	PositiveInteger	0..1	attr	This attribute defines how often a process shall be restarted if the start fails. <nowiki>numberOfRestartAttempts = "0" OR Attribute not existing, start once numberOfRestartAttempts = "1", start a second time </nowiki>
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. **Tags:**atp.Status=draft

▽

△

Class	**Process**			
stateDependent StartupConfig	StateDependentStartup Config	*	aggr	Applicable startup configurations. **Tags:**atp.Status=draft

Table A.4: Process