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	17-03

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2017-03-31	17-03	AUTOSAR Release Management	Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction and functional overview

This specification specifies the functionality of the AUTOSAR Functional Cluster Logging.

Adaptive Logging provides interfaces for applications to forward logging information onto the communication bus, the console, or to the file system. Every of the provided logging information has its own severity level.

For every severity level, a separate method is provided to be used by applications. (this also includes e.g. ARA::COM)

In addition, utility methods are provided to convert decimal values into the hexadecimal number system, or into the binary digit system.

To pack the provided logging information into a standardized delivering and representation 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 [2] PRS DLT protocol Specification.

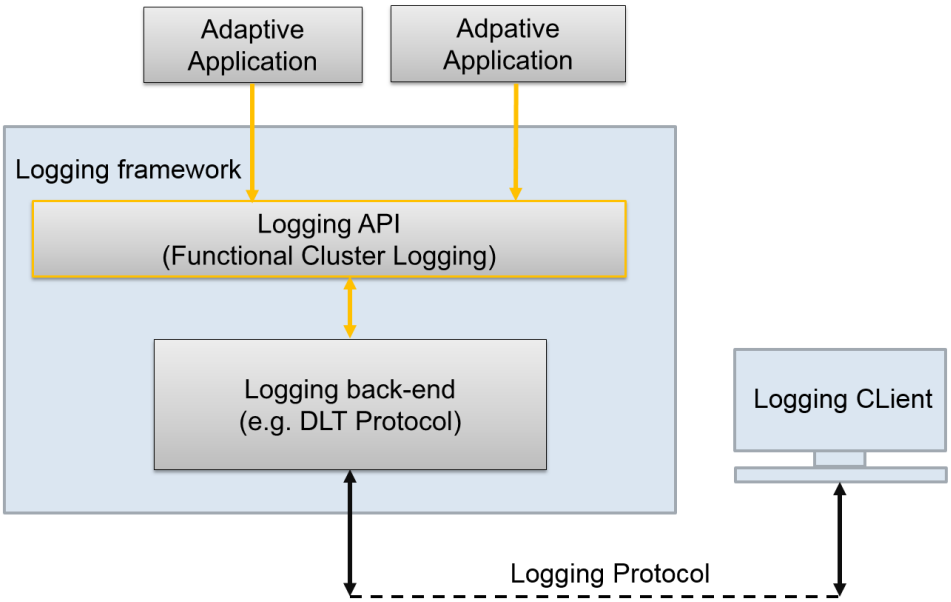

Figure 1: Architecture overview

2 Acronyms and abbreviations

Abbreviation / Acronym:	***Description:***
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 purpose
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] AUTOSAR Requirements on Logging
AUTOSAR_RS_Logging

[2] DLT protocol Specification
PRS_DLTProtocol

3.2 Related specification

N/A

4 Constraints and assumptions

4.1 Limitations

The provided C++ API is designed to be independent from the underlying Logging protocol back-end implementation.

4.2 Applicability to car domains

AUTOSAR Adaptive Log and Trace can be used for all car domains.

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

Requirement	Description	Satisfied by
RS_LOG_00001	Initialization and registration	SWS_LOG_00003, SWS_LOG_00004, SWS_LOG_00020
RS_LOG_00002	Meta information about Applications	SWS_LOG_00004, SWS_LOG_00020
RS_LOG_00003	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_00064, SWS_LOG_00065, SWS_LOG_00066, SWS_LOG_00067, SWS_LOG_00068, SWS_LOG_00069, SWS_LOG_0050, SWS_LOG_0051, SWS_LOG_0052, SWS_LOG_0062, SWS_LOG_0063
RS_LOG_00004	Grouping of Logging Information.	SWS_LOG_00005, SWS_LOG_00006, SWS_LOG_00021
RS_LOG_00005	Logging Information targets	SWS_LOG_00019
RS_LOG_00007	Provide raw buffer content	SWS_LOG_00014, SWS_LOG_00038, SWS_LOG_0061
RS_LOG_00008	Check the current severity level	SWS_LOG_00007, SWS_LOG_00070
RS_LOG_00009	Conversion functions for hexadecimal and binary values	SWS_LOG_00015, SWS_LOG_00016, 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_0053, SWS_LOG_0054, SWS_LOG_0055, SWS_LOG_0056, SWS_LOG_0057, SWS_LOG_0058, SWS_LOG_0059, SWS_LOG_0060
RS_LOG_00010	Early logging	SWS_LOG_00001, SWS_LOG_00002, SWS_LOG_00015, SWS_LOG_00017, SWS_LOG_00030, SWS_LOG_00031, SWS_LOG_00032, SWS_LOG_00033, SWS_LOG_00034, SWS_LOG_00035, SWS_LOG_00036, SWS_LOG_00037

7 Functional specification

This functional cluster specifies the usage of the defined C++11 API for logging purpose. Applications can use these functions to forward logging information to the bus, or to forward the logging information to the console or to the file system.

To do so, the following functionalities are provided:

1) Methods for initializing the Logging framework (see chapter 7.1)
2) Methods to initiate log messages (see chapter 7.2)
3) Utility methods to convert decimal values into hex- or binary values (see chapter 7.3)

7.1 Necessary parameters

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

The concept of log contexts:
In order to distinguish the logs from different logical groups within an application, every context within an application a particular ID and optionally a description has to be assigned.

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

A log context can be seen as a logger instance what is known from other on the market available logging frameworks. E.g. Android logger etc.

The using application needs to configure the Logging framework once at early startup with the following information:
- Application ID
- Application description
- The default log level
- The log mode
- The directory path

The using application can request contexts from the Logging framework, by providing the following information:
- Context ID
- Context description

7.1.1 Application ID

The Application ID is an identifier which 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, optionally an additional descriptive text can be provided. This Application description is passed as a string. The max. 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 7.3.

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 severity level will be processed by the Logging framework, the rest is ignored.

The Default Log Level is the programmatically 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 programmatically default log reporting level only application wide vs. having them also more fine granular on context level:

- Simplifying the API usage as much as needed.
- The more fine granular adjustments 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. Each of these destinations can be selected in parallel at the same time.

How and if the Logging API and its underlying protocol back-end supports these modes, is implementation detail.

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

Note:
Spend special attention to library components, which 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 own Context IDs system wide – at least when it's going to be used by more than one application.

7.1.7 Context description

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

7.2 Initialization of the Logging framework

Before logging information can be processed, the Logging framework needs to be initialized, and the using application need to be registered. To do so, information needs to be provided to the Logging framework like the Application ID and the log mode.

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 to. Possible destinations are the console, the file system, or the communication bus. These three destinations can also be used in combination at the same time.

Next to the registration of the applications at the Logging framework, also the so-called “contexts” need to be registered. These contexts are used to logically cluster logging information.

In addition to registration of the applications and the contexts, applications can verify the current active severity (also known as log level). To spare CPU load and memory consumption, this information can be used by the applications to avoid the generation of logging information, which will be filtered out by the Logging framework later on anyway.

[SWS_LOG_00001] ⌈
All messages logged before the initialization of the Logging framework is done shall be stored inside a ring-buffer with a limited size. That means, oldest entries are lost if the buffer exceeds. The size of the buffer is implementation detail.
⌋ (RS_LOG_00010)

[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) do not specify return values nor they throw exceptions.
⌋ (RS_LOG_00010)

[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_LOG_00001)

[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 (opt. in case of LogMode::kFile)

⌋ (RS_LOG_00001, RS_LOG_00002)

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 behind is, once a context was registered against the protocol back-end, its lifetime must be ensured until the end of application's process. ⌋ (RS_LOG_00004)

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

- The context ID
- The context description

⌋ (RS_LOG_00004)

Note:
This information might be used by the used Logging framework.

[SWS_LOG_00007] ⌈
Applications may want to check the actual configured reporting log level of certain logger contexts before doing log data preparation that is runtime intensive. To check if a desired log level is configured, the function `IsLogEnabled()` shall be called. Doing this avoids the CPU and memory consuming preparation of logging information, which will be filtered by the Logging framework later on anyway. ⌋ (RS_LOG_00008)

7.3 Log Messages

The Adaptive Logging framework offers stream based API for message creation that already supports all primitive data types (PoDs).

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's just another syntax, there is no difference compared to functions having a variadic argument pack. Actually compilers does expand them in the same way.

To initiate 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 severities are defined:

- Off (no logging)
- Fatal (Fatal error)
- Error (Error with impact to correct functionality)
- Warn (Warning if correct behavior cannot be ensured)
- Info (High level information)
- Debug (Detailed information for programmers)
- Verbose (Verbose debug message)

Note:
Off is not applicable for log message. This level can be used to set reporting level for the Logging framework either programmatically 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 to be processed log message is represented as a stream object which is an instances 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/destructor (e.g. empty ones) is cheap, actually instantiating an object in C++ equals of 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 feed 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 – when the variable goes out of scope, at the end of the function block.

Performance remark:
Due to the fact that no longer a LogStream object is 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 if depending 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 the developers need to take care for concurrent resource access as it is no longer ensured by the Logging API.

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_LOG_00003)

[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_LOG_00003)

[SWS_LOG_00010] ⌈
To initiate a log message with the Log level `Warning`, the API `LogWarning()` shall be called. This API returns LogStream object that has to be used by passing arguments via the insert stream operator “<<”.⌋
(RS_LOG_00003)

[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_LOG_00003)

[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_LOG_00003)

[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_LOG_00003)

[SWS_LOG_00014] ⌈
To log raw data by providing a buffer, the API `RawBuffer()` shall be called. ⌋
(RS_LOG_00007)

7.4 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 the hexadecimal or binary system, it shall be converted into the two's complement representation, whereas the most significant bit shall be set to '1' for negative numbers, and shall be set to '0' for positive numbers. ⌋ (RS_LOG_00009, RS_LOG_00010)

[SWS_LOG_00016] ⌈
To convert an integer decimal number into a hexadecimal format, the function `HexFormat()` shall be called. ⌋ (RS_LOG_00009)

[SWS_LOG_00017] ⌈
To convert an integer decimal number into a binary format, the API `BinFormat()` shall be called. ⌋ (RS_LOG_00010)

8 API specification

8.1 Type definitions

8.1.1 LogLevel

[SWS_LOG_00018] ⌈

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:	logcommon.hpp		
Description:	List of possible severity levels.		

⌋ (RS_LOG_00003)

8.1.2 LogMode

[SWS_LOG_00019] ⌈

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,
	kConole
	};		

Header file:	logcommon.hpp		
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.		

⌋ (RS_LOG_00005)

8.2 Function definitions

8.2.1 InitLogging

[SWS_LOG_00020] ⌈

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	

Description:	Initializes the logging framework for the application with given properties. In case that in logMode the kFile flag is set, the directory path needs to be provided as the. 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);
	}	

⌋ (RS_LOG_00001, RS_LOG_00002)

8.2.2 CreateLogger

[SWS_LOG_00021] ⌈

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	
Description:	Creates a Logger object, holding the context which is registered in the Logging framework.	

⌋ (RS_LOG_00004)

8.2.3 HexFormat (uint8)

[SWS_LOG_00022] conversion of a uint8 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format.	

⌋ (RS_LOG_00009)

8.2.4 HexFormat (int8)

[SWS_LOG_00023] conversion of an int8 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00009)

8.2.5 HexFormat (uint16)

[SWS_LOG_00024] conversion of a unit16 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format.	

⌋ (RS_LOG_00009)

8.2.6 HexFormat (int16)

[SWS_LOG_00025] conversion of an int16 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00009)

8.2.7 HexFormat (uint32)

[SWS_LOG_00026] conversion of a uint32 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format.	

⌋ (RS_LOG_00009)

8.2.8 HexFormat (int32)

[SWS_LOG_00027] conversion of an int32 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00009)

8.2.9 HexFormat (uint64)

[SWS_LOG_00028] conversion of a uint64 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format.	

⌋ (RS_LOG_00009)

8.2.10 HexFormat (int64)

[SWS_LOG_00029] conversion of an int64 into a hexadecimal value ⌈

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	
Description:	Logs decimal numbers in hexadecimal format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00009)

8.2.11 BinFormat (uint8)

[SWS_LOG_00030] conversion of a uint8 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format.	

⌋ (RS_LOG_00010)

8.2.12 BinFormat (int8)

[SWS_LOG_00031] conversion of an int8 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00010)

8.2.13 BinFormat (uint16)

[SWS_LOG_00032] conversion of a uint16 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format.	

⌋ (RS_LOG_00010)

8.2.14 BinFormat (int16)

[SWS_LOG_00033] conversion of an int16 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00010)

8.2.15 BinFormat (uint32)

[SWS_LOG_00034] conversion of a uint32 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format.	

⌋ (RS_LOG_00010)

8.2.16 BinFormat (int32)

[SWS_LOG_00035] conversion of an int32 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00010)

8.2.17 BinFormat (uint64)

[SWS_LOG_00036] conversion of a uint64 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format.	

⌋ (RS_LOG_00010)

8.2.18 BinFormat (int64)

[SWS_LOG_00037] conversion of an int64 into a binary value ⌈

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	
Description:	Logs decimal numbers in binary format. Negatives are represented in 2's complement.	

⌋ (RS_LOG_00010)

8.2.19 RawBuffer

[SWS_LOG_00038] ⌈

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	
Description:	Logs raw binary data by providing a buffer	

⌋ (RS_LOG_00007)

8.3 Class definitions

8.3.1 Class LogStream

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

Note:
Normally, using applications would not use this class directly, but use one of the log methods provided in the main logging API instead. Those methods automatically setup a temporary object of this class with the given log severity level. The only reason to get in touch with this class directly is, if developers want 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 Flush() method for further information.

Once this temporary object gets out of scope, its destructor is taking care that the message buffer is ready to be processed in the Logging framework.

8.3.1.1 Extending the Logging API to understand custom types

LogStream supports all of the PoDs natively. However, it can be easily extended for new complex 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] [

Service name:	LogStream::Flush
Syntax:	`void Flush ();`
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Exceptions:	None
Description:	Sends out the current log buffer and initiates a new message stream.

] (RS_LOG_00003)

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.

8.3.1.3 Build-in operators for natively supported types:

[SWS_LOG_00040] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00041] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00042] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00043] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00044] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00045] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00046] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00047] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_00048] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_0050] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_0051] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_0052] ⌈

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
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00003)

8.3.1.4 Build-in operators for conversion types:

[SWS_LOG_0053] [

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

] (RS_LOG_00009)

[SWS_LOG_0054] [

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

] (RS_LOG_00009)

[SWS_LOG_0055] [

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

] (RS_LOG_00009)

[SWS_LOG_0056] [

Service name:	LogHex64 handler
Syntax:	`LogStream& operator<<(const` LogHex64`& value) noexcept`
Parameters (in):	Reference to LogHex64 `value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends given value to the internal message buffer.

] (RS_LOG_00009)

[SWS_LOG_0057] ⌈

Service name:	LogBin8 handler
Syntax:	`LogStream& operator<<(const LogBin8& value) noexcept`
Parameters (in):	Reference to `LogBin8 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00009)

[SWS_LOG_0058] ⌈

Service name:	LogBin16 handler
Syntax:	`LogStream& operator<<(const LogBin16& value) noexcept`
Parameters (in):	Reference to `LogBin16 value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00009)

[SWS_LOG_0059] ⌈

Service name:	LogBin32 handler
Syntax:	`LogStream& operator<<(const LogBin32& value) noexcept`
Parameters (in):	Reference to LogBin32 `value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00009)

[SWS_LOG_0060] ⌈

Service name:	LogBin64 handler
Syntax:	`LogStream& operator<<(const LogBin64& value) noexcept`
Parameters (in):	Reference to LogBin64 `value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends given value to the internal message buffer.

⌋ (RS_LOG_00009)

8.3.1.5 Build-in operators for extra types:

[SWS_LOG_0061] ⌈

Service name:	LogRawBuffer handler
Syntax:	`LogStream& operator<<(const LogRawBuffer& value) noexcept`
Parameters (in):	Reference to `LogRawBuffer value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends plain binary data into message buffer.

⌋ (RS_LOG_00007)

[SWS_LOG_0062] ⌈

Service name:	std::string handler
Syntax:	`LogStream& operator<<(const std::string& value) noexcept`
Parameters (in):	Reference to `std::string value`
Parameters (inout):	None
Parameters (out):	None
Return value:	Reference to a `LogStream object`
Exceptions:	None
Description:	Appends STL string to message buffer.

⌋ (RS_LOG_00003)

[SWS_LOG_0063] ⌈

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

⌋ (RS_LOG_00003)

8.3.2 Class Logger

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

A context will be automatically registered against the LT 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. That means, using applications are not supposed to call the Logger constructor by themselves.

8.3.2.1 Logger::LogFatal

[SWS_LOG_00064] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.2 Logger::LogError

[SWS_LOG_00065] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.3 Logger::LogWarn

[SWS_LOG_00066] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.4 Logger::LogInfo

[SWS_LOG_00067] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.5 Logger::LogDebug

[SWS_LOG_00068] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.6 Logger::LogVerbose

[SWS_LOG_00069] ⌈

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

⌋ (RS_LOG_00003)

8.3.2.7 Logger::IsLogEnabled

[SWS_LOG_00070] ⌈

Service name:	Logger::IsLogEnabled

Syntax:	`bool IsLogEnabled (`
	`LogLevel logLevel`
	`) 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
Description:	The Application can check if the current configured log will pass desired log level.

⌋ (RS_LOG_00008)