Document Title	Demonstrator Design of Functional Cluster Log and Trace
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	861

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• Removed `InitLogging()` and implemented usage of `ara::core::Initialize` • Refactoring and editorial changes
2019-11-28	R19-11	AUTOSAR Release Management	• Adapt to API changes in SWS_LogAndTrace • Added Chapter 3 Usage example • Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Adapt to API changes in SWS_LogAndTrace • Set default log level of ara::log::CreateLogger to kVerbose (instead of kWarn) • Directory layout reorganization
2018-11-02	18-10	AUTOSAR Release Management	• Rename "logcommon.h" to "common.h" • Several symbols are hidden within ara::log::internal namespace • The header files no longer depend on DLT includes • Comments and doxygen tags clean-up

2018-05-02	18-03	AUTOSAR Release Management	• Minor changes
2017-11-22	17-10	AUTOSAR Release Management	• Minor changes
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document describes the design (approach and decisions) of the Functional Cluster Log and Trace for AUTOSAR's Adaptive Platform Demonstrator.

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply other implementations as the standard is defined by the specifications. Nevertheless, the demonstrator may supplement and ease the understanding of the specifications.

The main goal for development of the Log and Trace demonstrator code was to provide a proof of concept for the Functional Cluster features specified in AUTOSAR Adaptive.

This reference implementation does not realize the whole logging framework but utilizes the open source Genivi DLT-daemon and its API library as the logging communication protocol. That means, this demonstrator realizes a high-level abstraction API towards underlying back-end implementations, whereas further updates or even exchanging the whole back-end, won't influence the standardized public interfaces.

1.1 Known limitations

- The current version of underlying Genivi DLT is commit hash a961dba0013ed2119aa719546c63212459753549, which is a few commits ahead of v2.16.0. The used Genivi DLT does not fully conform to the AUTOSAR DLT protocol specification v4.3.

1.2 Deviations from specification

- The behavior of specification SWS_LOG_00019 is not fully supported in current state of implementation (see API documentation for details).

1.3 Used OSS components

This sections lists all OSS components used in the implementation of the Functional Cluster, their architectural context and the rationale for their use.

OSS	Version	Explanation	Binding	License	License Version
Genivi DLT daemon	2.16.0+X (see above)	DLT protocol implementation	Headers and API lib, dynamic linked	MPL	2.0
Google Test	1.8.0	Unit testing	Not relevant	3-Clause BSD	NA

2 Overview on architecture

2.1 Design approach / Design principles

The high-level logging API for AUTOSAR Adaptive Applications provides simple to use log methods and additional formatting utilities. It abstracts housekeeping work of used back-ends as much as possible in order to offer a convenient way of doing actual logging for developers. In addition to that, the API is designed to be easy extendable to enable developers to log own custom data types.

2.2 Overview on architecture

The public API consists of the following modules:

- Common Types (Definitions of data types used within the API namespace)
- Logger class (Represents a log context as defined in DLT protocol)
- LogStream class (Represents a log message that is associated with a log context)
- Main logging interface (Accumulation of various functions for the API usage)

The internal business logic of the Log and Trace Functional Cluster is designed to manage all necessary resources life time as well as the correct de-/registrations phases of the underlying back-end.

Besides that, the internal logic of the Log and Trace Functional Cluster will automatically create a default log context on demand, based on lazy-init, if an using application did not explicitly create any own log contexts. This design decision is based on the approach of doing logging "easy as much as possible". So, if an application has only trivial requirements for its logging, it can go ahead of using the convenience functions only, without dealing with the details of Logger and LogStream class.

In order to achieve this, an internal core component was implemented (the LogManager class) that exists once per application instance, to be more precise, once per OS process. This singleton of the LogManager class will be setup as soon as the using application has called the InitLogging function.

Since the LogManager realizes internal behavior, it is not part of the public API.

2.3 Class Overview

2.3.1 Logger

Class representing a DLT logger context. DLT protocol defines so called "contexts" which can be seen as logger instances within one application or process scope.

Refer to the Logger class API chapter, for further detailed description.

2.3.2 LogStream

Class representing a log message. A log message is always associated to a certain log context. To append data (log arguments) to the message, insert stream operators are to be used.

All of the plain old data types are supported out of the box, but it can be extended to understand new/custom data types, simply by providing own operator<< implementations.

Refer to the LogStream class API chapter, for further detailed description.

3 Usage example

The following minimal sample code demonstrates the usage of the implementation of Log and Trace:

```
#include "ara/log/logging.h"
#include "ara/log/log_stream.h"

using namespace ara::log;

int main(int argc, char **argv) {

/*
 * Explicitly configure the logging framework.

This will be handled by ara:core::Initialize() in future.
Specific work-around for ara-demonstrator-platform.
 * ************************************************************
 */
InitLogging(
            "TEST",
            "This is a test application",
            LogLevel::kInfo,
            LogMode::kRemote | LogMode::kConsole,
            "/tmp/");

Logger& context1 = CreateLogger("CTX1", "A sample context", LogLevel::kInfo
    );

context1.LogInfo()
        << (bool) true
        << (uint16_t) 1000 /* uint8_t, uint_32 and uint_64 defined as well
    */
        << (int16_t) -1000 /* int8_t, int_32 and int_64 defined as well */
        << (float) -1.2345 /* double defined as well */
        << HexFormat((uint8_t)0xAB) /* uint_16/32/64 defined as well */
```

```
30          << BinFormat((uint8_t)0xCD) /* uint_16/32/64 defined as well */
31          << "TestString"     /* ara::core::StringView defined as well */;
32
33 return 0;
```

In line 12-17, the underlying logging framework DLT is configured. While this information will be supplied via the manifest file in future, application ID, application description, default log level, log mode and file path for logging to file are specified explicitly this way. When the application as such is defined, it needs one or more logging contexts. A logging context is defined by a context ID, a context description and a loglevel as depicted in in line 20.

Each log context owns six member functions to log messages, one for each log level: LogFatal(), LogError(), LogWarn(), LogInfo(), LogDebug() and LogVerbose(). They return an object of type LogStream. Its usage is depicted in line 22-29.

The example above results in console output similar to:

{{

[...] ECU1 TEST CTX1 log info V 7 [1 1000 -1000 -1.2345 0xab 0b1100 1101 Test String]

}}