
ibeo
automotive

ibeo automotive

Ibeo SDK - Logging Services

User Manual

Ibeo Automotive Systems GmbH
Merkurring 60-62
D - 22143 Hamburg
Phone: +49 - (0) 40 298 676 - 0
Fax: +49 - (0) 40 298 676 - 10
E-mail: info@ibeo-as.com

Version History

Date	Version	Changes
20-Feb-2019	1.0	Initial Version of this manual

Table 0.1: Version History

Contents

1 Introduction

The ibeoSDK Logging Services are used to output log statements that aid debugging the code. It may be the only way of getting information about the status of the application e.g. when it is not possible to use a debugger. These services are not only used inside the SDK itself but are open to applications too. Please refer to section 2-API for how to use the Logging Services API.

The following picture shows the general architecture of the Ibeo SDK Logging Services:

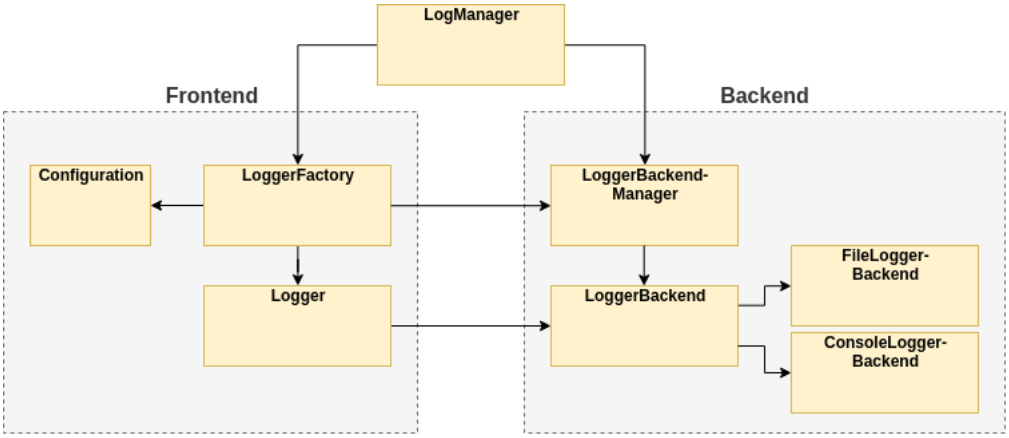

Figure 1.1: Architecture

An application uses Logger instances from the frontend to send messages of a given severity defined by the LogLevel to the logging system. To create a logger, the application chooses a unique identifier which is formatted like a C++ class name with namespace: e.g. `ibeo::common::logging::Logger` and calls the `createLogger` method from the `LogManager` class. The number of loggers an application can create is not limited by the Logging Services. So, it is possible to use a separate logger for each class or even for each class instance (e.g. if the class name is suffixed by an index or ID). The more loggers an application uses, the finer the logging can be controlled (see section 3-Configuration).
A special logger is the global logger which can be retrieved using the `LogManager`'s `globalLogger` method. This instance can be used to quickly log messages without worrying about creating a logger. The downside is, that only one log level can be set for all messages sent through this logger.

The backend part of the logging system receives the log messages, filters them according to the currently set log level, and processes them. There are predefined logging backends for printing the log messages to the console (`ConsoleLoggerBackend`) or to file (`FileLoggerBackend`). Each backend is identified by a unique class ID. An application can create additional backends (e.g. for writing log messages into GUI window or a database). This is covered in section 4-Writing Custom Backends.

2 API

2.1 Prerequisites

The ibeoSDK Logging Services consist of quite a number of classes with their corresponding header files. For convenience, there is a single include file `ibeo/common/logging/logging.hpp` that contains all necessary includes and some macro definitions (see section 2.4-Sending Log Messages).
All classes are defined in the C++ namespace `ibeo::common::logging`. To use them, they should be referenced by their fully qualified name (e.g. `ibeo::common::logging::LogManager`), by adding a using-directive (e.g. `using namespace ibeo::common::logging`), or with a namespace-alias-definition
(e.g. `namespace logging=ibeo::common::logging`). In the latter case, the corresponding class must be referenced through that alias (e.g. `logging::LogManager`).

2.2 Obtaining a Logger

The first task for a developer when writing log messages is to get a logger from the Logging Services. This can be achieved in two different ways:

- by using the Log Manager instance for creating a logger with a unique ID

  ```
  LogManager& logManager=LogManager::getInstance();
  LoggerSPtr logger=logManager.createLogger("ibeo::common::app::Foo");
  ```

  Although the logger ID can be any string, it is recommended to use the name of the class (together with its namespace) that creates the logger. This makes configuring the Logging Service easier (see section 3-Configuration).

- or by using the global logger

  ```
  LoggerSPtr logger = LogManager::globalLogger();
  ```

  The global logger is a predefined instance in the Logging Services with the ID `ibeo::common::logging::GlobalLogger`.

2.3 Log Levels

The Logging Services define the following log levels which are used to define the severity of log messages.

- *Critical* messages should be used for very important issues where it is most likely that the application is not running correctly any longer.
- *Error* is for severe messages where it is likely that the application does not work as expected any more.
- *Warning* messages indicate errors that have been recovered. It is most likely that the application still runs as expected.
- *Info* is for general purpose information, e.g. about the application's status.

- *Trace* messages should be used to log communication with external systems (e.g. a network or a database connection).
- *Debug* is for all other messages helping debugging the code.

The log levels are listed from most to less important. If - for instance - the level of a logger is set to *Info* in the configuration all messages with levels *Critical*, *Error*, *Warning*, and *Info* are forwarded to the backend while messages with levels *Trace* and *Debug* are filtered out. Messages with level *Critical* cannot be disabled.

2.4 Sending Log Messages

Once a logger is obtained, it can be used to send log information by creating a `LogMessage` object and sending it to the backend with the logger's `log` function.

```
MessageSPtr msg = std::make_shared<ibeo::common::logging::Message>
                  (__LINE__, __FUNCTION__, __FILE__, "Log text.");
log(LogLevel::Info, msg);
```

The `Message` object carries information about the name, file path, and line number of the function where the message is created (represented by the predefined C++ macros in the example above).
The `LOGMSG\_TEXT` macro can be used to shorten the creation of log messages:

```
log(LogLevel::Info, LOGMSG_TEXT("Log text."));
```

The log message text can be extended by using the stream output operator "«" and IO manipulators. In this case the `LOGMSG` macro should be used instead:

```
log(LogLevel::Info, LOGMSG << "Magic number: 0x" << std::hex << std::setw(4)
                           << magicNumber);
```

There are also convenience functions for each log level, e.g.:

```
logger->warning(LOGMSG << "Log text.");
```

As an alternative, the following macros can be used:

```
LOGCRITICAL(logger, LOGMSG << "Log text.");
LOGERROR(logger, LOGMSG << "Log text.");
LOGWARNING(logger, LOGMSG << "Log text.");
LOGINFO(logger, LOGMSG << "Log text.");
LOGTRACE(logger, LOGMSG << "Log text.");
LOGDEBUG(logger, LOGMSG << "Log text.");
```

3 Configuration

The configuration of the logging system can be either be set programmatically or loaded from XML files. The latter is the preferred method because this allows to change the configuration without re-compiling the application.
At startup, the Logging Services will automatically load an XML configuration file named `logconfig.xml` from the current working directory. If that file cannot be found, the system falls back to the default configuration that is:

- The log level is set to *Error* for all loggers.
- All loggers print their messages to console only.

3.1 XML file based Configuration

The XML configuration file (see section A.1-Sample XML Configuration File) is made of two sections: *Backends* and *Loggers*.

3.1.1 Backend Configuration

The *Backends* section contains a list of *Backend* XML elements that configure the individual backends. Each backend is identified by the XML attribute *id* which is defined in the `Backend` class. The backends currently provided by the Logging Services are:

- `ConsoleLoggerBackend` ("ibeo::common::logging::ConsoleLoggerBackend") for printing messages to the console. Levels *Critical* and *Error* are printed to stderr, other levels to stdout.
- `FileLoggerBackend` ("ibeo::common::logging::FileLoggerBackend") for storing the messages in a file.

Each backend configuration also has a *format* XML attribute describing how the log messages are formatted (see section 3.3-Formatting for a detailed format description). This allows different formatting for each backend, e.g. printing less details on the console, but more details in the log file.
Other backend parameters are specified as inner XML elements and are defined by the backend itself, like the *Path* for the *FileLoggerBackend*.
The Logging Services can be extended by additional backends that are also configured under the *Backends* section (see section 4-Writing Custom Backends).

3.1.2 Logger Configuration

A logger configuration is specified in a *Logger* XML element under the *Loggers* section and is identified by the XML attribute *id* which has to be specified when a logger is created in the application (see section 2.2-Obtaining a Logger). To confiugure a group of loggers identically, the "*" can be used in the logger ID. E.g. the configuration with ID "ibeo::common::logging::*" will be valid for all loggers whose ID starts with "ibeo::common::logging::", i.e. all loggers defined in the Logging Services itself. The other XML attribute *level* sets the log level of the corresponding logger (see section 2.3-Log Levels).
The inner XML element *BackendRef* is used to connect a logger with a backend which is identified by the *id* XML attribute. This e.g., allows certain loggers to log on the console only while others log to both console and file.

A special logger in the *Loggers* section is the *Root* logger. The configuration parameters are the same as for the other loggers, except for the *id* attribute which is not necessary and form a default configuration that is used for all loggers that do not have a separate entry under the *Loggers* section.

3.2 Programmatic Configuration

Apart from the automatic loading of the XML configuration file (see section 3-Configuration), there are multiple ways for configuring the Logging Services programmatically. An application can

- instruct the Logging Services to use a different XML file by calling the `LogManager`'s `loadConfig` method,
- create an XML formatted string with the configuration in memory and call the `LogManager`'s `parseConfig` method, or
- construct a `Configuration` object and call the `LogManager`'s `configure` method.

3.3 Formatting

The conversion of a log message into a textual output is determined by a format string that is set in the configuration. This string consists of conversion patterns which are replaced by the corresponding log message field and plain text in any order and multiplicity. Each conversion pattern can have alignment and length parameters, some can be extended with optional arguments enclosed in brackets "". The following table list all conversion pattern.

Conversion Pattern	Purpose	Arguments
%level	Print the log level.	*length*: used to limit the length of the output (optional, default is no limit). *lowerCase*: used to convert the output to lower case, if set to true, or upper case otherwise (optional, default is upper case). Example: %level{length=1}{lowerCase=true}
%date	Print the date and/or time when the log message was created.	String with date/time format. Possible values are: • DEFAULT: use default format ("%Y-%m-%d %H:%M:%S,%s") • UNIX: print the number of seconds since start of epoch (1970-01-01 00:00:00). • UNIX_MILLIS: print the number of milliseconds since start of epoch (1970-01-01 00:00:00). • Any valid format string that can be interpreted by the strftime function. As a special extension the parameter %s (lower case character) can be used here to print the milliseconds. Example: %date{"Now it's %T,%s."}

Conversion Pattern	Purpose	Arguments
%file	Print the path of the source file where the log message was generated.	Number limiting the count of elements to be printed. Truncation is done from the beginning of the file path. Example: %file{2} (if used with a file path "foo1/foo2/bar.cpp" it will print "foo2/bar.cpp").
%func	Print the name of the function where the log message was generated.	None. Example: %func
%func	Print the line number in the source file where the log message was generated.	None. Example: %line
%msg or %message	Print the log message text.	None. Example: %msg
%logger	Print the ID of the logger that created the log message.	Number limiting the count of elements to be printed. Truncation is done from the beginning of the logger ID. Example: %logger{2} (if used with a logger ID "ibeo::ref::tools::app::Foo" it will print "app::Foo").
%seqNo	Print the sequence number of the log message. This global number is incremented every time a new log message is created . Thus, it can be used to uniquely identify a log message int the output and for sorting.	None. Example: %seqNo
%thread	Print the ID of the thread that executes the code that sent the log message. *Note: the output may vary across different operating systems.*	None. Example: %thread

Each conversion pattern can have an optional alignment and/or length modifier located between the percent sign and the conversion pattern name. Possible formats are:

%15msg	the output is at least 15 characters wide and right aligned, e.g. "·····Log text."
%-15msg	the output is at least 15 characters wide and left aligned, e.g. "Log text.·····"
%.5msg	the output is at most 5 characters wide and truncated from beginning, e.g. "text."
%.-5msg	the output is at most 5 characters wide and truncated from end, e.g. "Log t"
%10.15msg	the output is at least 10 characters and at most 15 characters wide, right aligned if shorter and truncated from beginning if longer
%-10.15msg	the output is at least 10 characters and at most 15 characters wide, left aligned if shorter and truncated from beginning if longer
%10.-15msg	the output is at least 10 characters and at most 15 characters wide, right aligned if shorter and truncated from end if longer
%-10.-15msg	the output is at least 10 characters and at most 15 characters wide, left aligned if shorter and truncated from end if longer

4 Writing Custom Backends

Sometimes the provided backends are not sufficient, e.g. when logging into a database or a message window of a GUI application. Therefore, the logging system provides interfaces for writing custom backends.

4.1 Creating a Custom Backend

Backends can operate in one of two modes:

- A synchronous backend handles the log messages immediately after they were sent to the logging system. As this task runs in the same thread, the application is typically blocked during this time. Thus, this method is good when you need immediate output, but should not be used for longer processing times.
  To implement a synchronous backend create a class that derives from `LoggerBackend` and override the `log` method.
- An asynchronous backend uses a queue internally to decouple the processing of the log messages in the backend. Thus, the application is not blocked when the log message is sent and the processing is done in a separate thread. This is the preferred method when the processing takes a noticable time, e.g. when sending log messages to am remote database.
  To implement an asynchronous backend create a class that derives from `AsyncLoggerBackend` and override the `logAsync` method.

4.2 Register a Custom Backend

To let the logging system know about an additional backend the method `LogManager::registerBackend` should be called with an instance of the new backend.

NOTE

All custom backends must be registered before the configuration is loaded!
Otherwise, the custom backends might not work as expected.

4.3 Configuring a Custom Backend

The configuration of a custom backend is placed in the configuration file in the same way as for the provided backends (see 3-Configuration). The logging system takes care that the corresponding section in the XML file is given to the `configure` method in the custom backend. The content of this section is transparent to the logging system. So, any parameter definition can be placed here as long it adheres to the XML formatting rules. E.g.:

```
<Backend id="ibeo::common::database::DbLoggerBackend">
    <ServerName>db.ibeo.as</ServerName>
    <UserName>user</UserName>
    <Password>password</Password>
    <Instance>myDatabase</Instance>
</Backend>
```

A Appendix

A.1 Sample XML Configuration File

```
<?xml version='1.0' encoding='UTF-8'?>
<Configuration>
    <Backends>
        <Backend
            id="ibeo::common::logging::ConsoleLoggerBackend"
            format="%d [%t] %-5level %.30logger - %msg%n" />
        <Backend
            id="ibeo::common::logging::FileLoggerBackend"
            format="%d [%t] %-5level %.30logger - %msg%n">
            <Path>ibeo.log</Path>
        </Backend>
    </Backends>

    <Loggers>
        <Root level="debug">
            <BackendRef
                id="ibeo::common::logging::ConsoleLoggerBackend" />
            <BackendRef
                id="ibeo::common::logging::FileLoggerBackend" />
        </Root>
        <Logger
            id="ibeo::common::app::Foo"
            level="critical">
            <BackendRef
                id="ibeo::common::logging::ConsoleLoggerBackend" />
        </Logger>
    </Loggers>
</Configuration>
```