AS Logger

Structured Logging and Alarm Recording for CODESYS PLC Applications

Title	AS Logger
Document Type	User Manual
File Reference	AS_Logger.library
Version	2.2
Date	17.08.2026
Author	ASKS GmbH
Status	Released

ASKS GmbH

Tevesstraße 29a · 78176 Blumberg · www.as-ks.de · info@as-ks.de

Table of Contents

Summary: This user manual describes the ASKS Logger – a CODESYS library for structured logging and alarm recording. It covers integration, initialization, decoupled writing via the inbox, and the configuration and registration of sinks (incl. FileSink rotation) and decorators, filtering, alarm logging, error codes, and visualization.

1 Overview

The ASKS Logger (AS_Logger) provides structured logging and alarm recording for CODESYS PLC applications. A log event is created via a unified API, optionally enriched by decorators, evaluated by filters, and distributed to one or more output destinations (sinks).
The library's key features include:

- Six log levels: TRACE, DEBUG, INFO, WARN, ERROR, FATAL
- Chainable fluent API with structured key/value pairs and placeholder templates
- Alarm logging with severity levels and alarm codes
- Multiple simultaneous sinks: memory buffer for visualization, CODESYS log (as a standard component or as its own log instance), files, and syslog per RFC 5424 over UDP
- Automatic event enrichment: machine/host context, source context, masking of sensitive fields
- Per-sink filtering
- Open interfaces (ISink, IDecorator, ISinkFilter) for custom extensions without touching the core

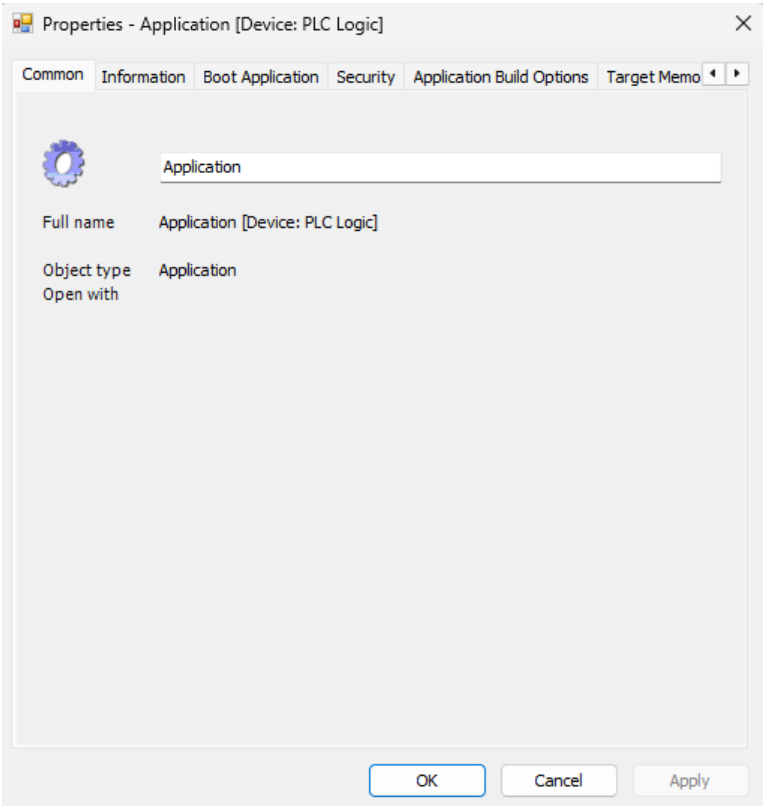
Writing log events is decoupled from output: a log call simply places the event into an inbox and returns immediately. A dedicated worker task, created by the logger itself, processes the inbox — no cyclic call from the application is required. The inbox holds 100 entries by default; on overflow the oldest entries are discarded (ring-buffer behavior).
The library requires the AS_Util library (UTL) (e.g., Semaphore, system string parameters).

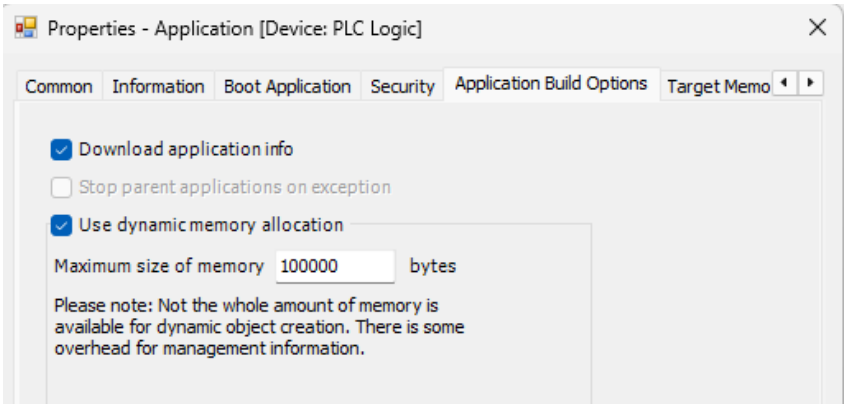
2 Instructions

2.1 Application

To use the logger functions, the AS_Logger library must be added in the Library Manager. The AS_Util library (UTL) is required as a dependency.

The logger uses dynamic memory, so the application's dynamic memory must be increased: under Application → Properties → Application Build Options, enable the "Use dynamic memory allocation" option and increase the maximum size.





Note: Without extending the dynamic memory, the logger cannot be initialized.

2.2 Logger

A local logger instance is declared via the type LOC.Logger. Within a local instance, sinks, decorators, and filters are addressed without a namespace prefix. If the instance is registered as a global reference, access from anywhere is via LOC.UTL.Logger (static calls) or LOC.UTL.logger (fluent calls).
The minimum-level check happens before the event is built, so events below the configured level incur only minimal overhead.

```
// --- Minimum level ---

// Set global minimum log level
eErr := LOC.UTL.logger.SetMinLevel(eLevel := LOC.UTL.LOG_LEVEL.DEBUG);
```

Writing is decoupled from output: a log call simply places the event into the logger's inbox and returns immediately. A dedicated worker task, created by the logger itself, processes the inbox — no cyclic call from the application is required, and slow sinks (e.g., syslog over UDP) never block the calling task.

2.2.1 Global Logger Reference

The local logger instance can be registered once as a global reference, so the logger is reachable from any POU without passing the instance around:

```
// === Logger ===

lo   : LOC.Logger;
```

```
// --- Set global logger reference ---

// Set local Logger as global Logger reference
LOC.UTL.Logger.SetLogger(lo);
```

After registration, every further access — configuring sinks, decorators, and filters, as well as all log and alarm calls — goes through LOC.UTL.Logger (static calls, e.g., SetLogger, diagnostic counters) or LOC.UTL.logger (fluent calls such as SetKey, Source, Info, LogAlarm). The local instance is no longer used directly after that.

2.2.2 Structured Logging and Placeholders

Key/value pairs passed via SetKey are inserted into matching {key} placeholders in the message template. Source sets the event source. The level methods (Trace, Debug, Info, Warn, Err, Fatal) return a reference to the logger and are therefore chainable.

```
LOC.UTL.logger
    .SetKey(sKey := 'speed', sVal := '450')
    .SetKey(sKey := 'time',  sVal := '00:12:30')
    .Source(sSource := __POUNAME())
    .Info(sTemplate := 'Granulation running, mixer speed {speed} rpm, elapsed
{time}');
```

If a sequence should not be interrupted by other tasks, the chain is wrapped in Lock/Unlock:

```
LOC.UTL.logger
    .Lock()
    .SetKey(sKey := 'session', sVal := 'sess-batch-88')
    .SetKey(sKey := 'action',  sVal := 'emergency_stop')
    .Source(sSource := __POUNAME())
    .Err(sTemplate := 'Critical fault in session {session} during {action}')
    .Unlock();
```

2.3 Sinks

A sink is an output destination that implements the ISink interface. Sinks inherit from SinkBase, which applies the sink's own minimum level, the assigned filter, and local decorators before the actual output (DoEmit). Each sink is configured independently via AddDecorator, SetFilter, and SetMinLevel.

Sink	Purpose
MemorySink	Ring buffer in memory for display/visualization (GetCount, GetAt, UpdateVisu)
CodesysLogSink	Writes entries as its own component into the standard CODESYS log (SetComponent defines the component name; the log level is mapped to the matching CODESYS log class)
CmpLogSink	Creates its own, new CODESYS log instance (instead of writing into a component of the standard CODESYS log) and writes entries there. The instance can be selected in the log viewer via the dropdown field at the top right (Init).
SyslogUdpSink	Syslog per RFC 5424 over UDP (Init with server IP/port, hostname, application name, facility, PEN)
FileSink	Writes events as logfmt lines to a file on the controller via AS_File — no runtime log component required. Size-based rotation with configurable file size and archive count (Init).

Table 1: Available sinks

```
// === Sinks ===
```

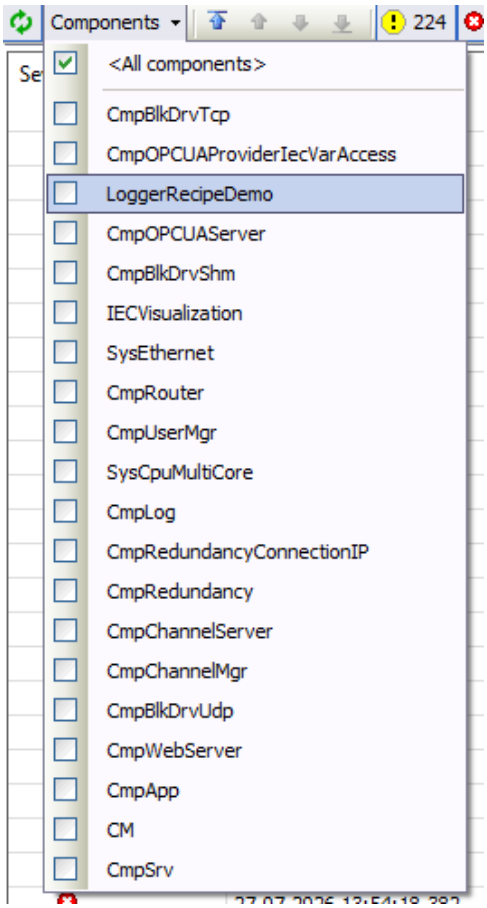
```
plcLogSink    : CodesysLogSink;
memSink       : MemorySink;
filesink      : FileSink;
syslogSink    : SyslogUdpSink;
CmpLogSink    : CmpLogSink;
```

```
// --- Sink configuration ---

// Create new component in CODESYS log
eErr := plcLogSink.SetComponent(sComponent := 'LoggerRecipeDemo');

eErr := syslogSink.Init(
            // Init UDP sink
            sServerIp   := '192.168.100.77',
            // Local network adapter IP used to send
            sLocalIp    := '192.168.100.111',
            // Hostname reported in the syslog message
            sHostname   := 'PLC01',
            // Application name reported in the syslog message
            sAppName    := 'LoggerRecipeDemo');

eErr := filesink.Init(
            // Name of the file (base name, without extension)
            sName       := 'Test',
            // Directory to store files in
            sDir        := 'PlcLog',
            // File type / extension
            sFileType   := 'txt',
            // Max size per file in bytes (1 MB)
            nMaxBytes   := 1048576,
            // Max number of rotated files
            nMaxFiles   := 5);

eErr := CmpLogSink.Init(
            // Create new Logger instance in CODESYS Device Log
            sName       := 'TestCMP',
            // Name of the component
            sComponent  := 'LoggerDemo');

// Number of visible MemorySink lines
eErr := memSink.Init(nVisuLines := 30);
```

2.3.1 CodesysLogSink

SetComponent() creates its own component in the standard CODESYS log, under which the entries appear.

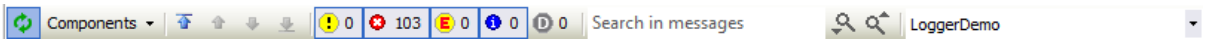
```
// Create new component in CODESYS log
eErr := plcLogSink.SetComponent(sComponent := 'LoggerRecipeDemo');
```



2.3.2 CmpLogSink

Unlike CodesysLogSink, CmpLogSink creates its own, new log instance, which can be selected in the log viewer via the dropdown field at the top right.

```
eErr := CmpLogSink.Init(
            // Create new Logger instance in CODESYS Device Log
            sName       := 'LoggerDemo',
            // Name of the component
            sComponent  := 'AppLog');
```



2.3.3 FileSink

FileSink writes each event as a logfmt line to a file in the controller's file system (via FIL.FileAscii) and — unlike CodesysLogSink/CmpLogSink — requires no runtime log component (CmpCodesysLog):

```
2026-07-23-12:34:56.789 INFO Belt started user=jj pump=3
```

Init() creates the target directory if needed and sets the file name, type, and rotation limits. If sDir or sName is empty, Init returns ERR_WRONG_INPUT.
Size-based rotation: before each write, FileSink checks whether the current file size (read from FIL.FileAscii.nSize, correct even after a restart) plus the new line would exceed nMaxBytes. If so, it rotates first: the oldest archive file (index nMaxFiles) is deleted, all remaining archives are renumbered (.i → .(i+1)), and the current file becomes archive index 1.
Archive files are named following the pattern <Name>_<Index>.<Type>, e.g. Test_1.txt, Test_2.txt ... up to Test_5.txt for nMaxFiles := 5. The file stays open between writes and is only closed on rotation or in FB_Exit. File I/O errors are reported as ERR_LOG_RUNTIME_FAILURE.

2.3.4 Registering Sinks

A configured sink only becomes active once it is additionally registered with the logger — without this step, Init, SetFilter, and similar calls have no effect, since the sink never receives an event from Emit:

```
// --- Register all sinks ---

// Register CODESYS log sink
eErr := LOC.UTL.logger.AddSink(sink := plcLogSink);
// Register memory sink
eErr := LOC.UTL.logger.AddSink(sink := memSink);
// Register file sink
eErr := LOC.UTL.logger.AddSink(sink := filesink);
// Register cmp sink
eErr := LOC.UTL.logger.AddSink(sink := CmpLogSink);
```

AddSink returns ERR_LOG_SINK_LIMIT_REACHED if the maximum number of sinks (GV_LOG.c_nMaxSinks) has already been reached.

2.4 Decorators

A decorator enriches an event before output (IDecorator interface). Decorators can be added globally at the logger (logger.AddDecorator) or per sink (sink.AddDecorator). A sink-local decorator operates on a copy of the event and does not affect other sinks.

Decorator	Purpose
MachineDecorator	Adds machine/host context — hostname, serial number, firmware version, location — via SetConfig
SourceContextDecorator	Adds a freely definable source context via SetSource (e.g., to identify the origin of individual sinks)

RedactionDecorator	Masks sensitive keys in the payload via AddKey, using a configurable placeholder (SetReplacement)

Table 2: Available decorators

Example: mask sensitive keys only on the syslog sink — the console and memory sink show the real values, syslog shows "[***]".

```
Operator [***] (badge [***]) acknowledged alarm, credentials [***] verified
```

As with sinks, registration is what makes decorators effective — globally at the logger for all sinks, or locally on a single sink:

```
// --- Global decorators ---

// Set decorator for all sinks
eErr := LOC.UTL.logger.AddDecorator(decorator := machineD);

// --- Sink-local decorator ---

// Set decorator for FileSink
eErr := filesink.AddDecorator(decorator := redactor);
```

Global decorators run before sink-local ones. AddDecorator returns ERR_LOG_DECORATOR_LIMIT_REACHED (globally limited by GV_LOG.c_nMaxGlobalDec) once the respective limit is reached.
RedactionDecorator.SetReplacement defaults to [REDACTED] unless a custom placeholder is set (changed to [***] in the example above). SourceContextDecorator.SetSource becomes a no-op when passed an empty string (no source tagging).

2.5 Filters

A filter decides whether an event is accepted (ISinkFilter interface, Accept returns BOOL). Filters are configured per sink via SetFilter.

Filter	Purpose
LevelFilter	Allows events within a minimum/maximum level (SetMinLevel, SetMaxLevel)
AlarmSeverityFilter	Accepts/rejects based on a severity mask; can require an alarm property (SetSeverityMask, AcceptSeverity, RejectSeverity, SetRequireAlarmProperty, Clear)

Table 3: Available filters

LOC.UTL.ALARM_SEVERITY is built as a bitmask (NONE=1, INFO=2, WARN=4, STOP=8, ABORT=16, REBOOT=32), so any combination of severities can be represented as an OR combination. AcceptSeverity/RejectSeverity enable or exclude individual severities, while SetSeverityMask sets the entire mask in one call. SetRequireAlarmProperty(xRequire := TRUE) discards all entries without an alarm severity — useful for sinks that should carry alarms only. LevelFilter:

```
// --- Sink Level Filter: only info and above ---

// Set min level to INFO and above
eErr := levelFilter.SetMinLevel(eLevel := LOC.UTL.LOG_LEVEL.INFO);
// Set filter for MemorySink
eErr := memSink.SetFilter(filter := levelFilter);
// Set filter for FileSink
eErr := filesink.SetFilter(filter := levelFilter);
```

AlarmFilter:

```
// --- Sink Alarm Filter: ---

eErr := alarmFilter.AcceptSeverity(eSeverity := LOC.UTL.ALARM_SEVERITY.STOP);
eErr := alarmFilter.AcceptSeverity(eSeverity := LOC.UTL.ALARM_SEVERITY.ABORT);
eErr := alarmFilter.AcceptSeverity(eSeverity := LOC.UTL.ALARM_SEVERITY.REBOOT);
// Set filter for CmpLogSink
eErr := CmpLogSink.SetFilter(filter := alarmFilter);
```

2.6 Alarms

Alarms are logged with LogAlarm; the arguments are a LOC.UTL.ALARM_SEVERITY, an alarm code, and a message template. The helper program AlarmMapping provides conversions between alarm severity and log level (ToLogLevel, StringToSeverity).

```
LOC.UTL.logger
    .SetKey(sKey := 'temp', sVal := '132.8')
    .Source(sSource := __POUNAME())
    .LogAlarm(
        eSeverity  := LOC.UTL.ALARM_SEVERITY.STOP,
        nAlarmCode := 3001,
        sTemplate  := 'Sterilization temp {temp}°C exceeds critical max, aborting
cycle'
    );
```

2.7 Diagnostics

The logger provides diagnostic counters: nTotalEvents (all events created), nDroppedByMinLevel (discarded by the minimum-level check), and nWritten (events actually written). ResetCounters resets the counters.

```
// --- Diagnostics ---

// Current event counter
nEvents         := LOC.UTL.Logger.nTotalEvents;
// Dropped events
nDroppedEvents  := LOC.UTL.Logger.nDroppedByMinLevel;
// Actual written events
nWritten        := LOC.UTL.Logger.nWritten;
// Reset all counters
LOC.UTL.Logger.ResetCounters();
```

2.8 Error Codes

All methods return a value of type LOC.ERROR. In addition to general base codes (including ERR_ALREADY_INIT, ERR_WRONG_INPUT, ERR_NOT_FOUND, ERR_OVERFLOW), AS_Logger defines the following specific codes:

Code	Meaning
ERR_LOG_SINK_LIMIT_REACHED	Maximum number of sinks (GV_LOG.c_nMaxSinks) already registered via AddSink
ERR_LOG_DECORATOR_LIMIT_REACHED	Maximum number of decorators (global or sink-local) already registered via AddDecorator
ERR_LOG_PROP_LIMIT_REACHED	Maximum number of key/value pairs (SetKey) for an event reached
ERR_LOG_REDACTION_KEY_LIMIT_REACHED	Maximum number of masked keys in the RedactionDecorator (AddKey) reached
ERR_LOG_LEVEL_BELOW_MIN	Event is below the configured minimum level, discarded by Emit
ERR_LOG_FILTER_REJECTED	Event rejected by the assigned filter (SetFilter)
ERR_LOG_NULL_REFERENCE	Sink not initialized (e.g., FileSink without a prior Init call)
ERR_LOG_RUNTIME_FAILURE	Runtime error during output, e.g., file I/O error in FileSink
ERR_LOG_CFG_LOGGER_MISSING	Access via LOC.UTL.Logger before SetLogger has registered a local instance as the global reference

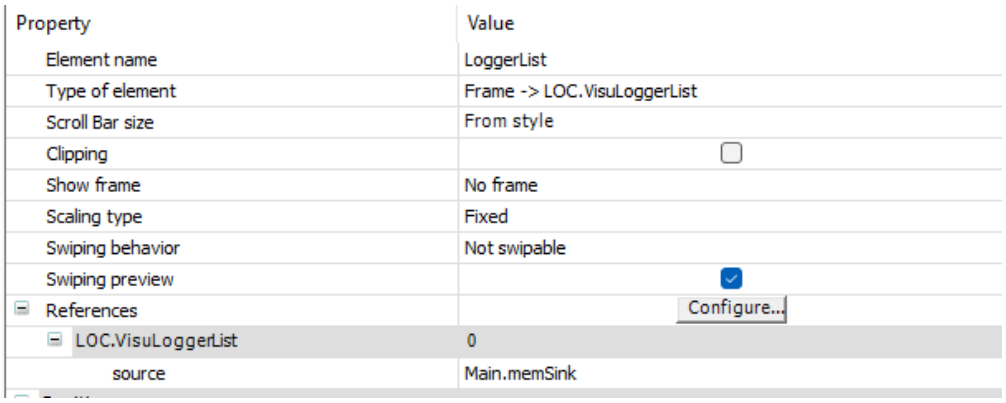
Table 4: Error codes (LOC.ERROR)

3 Visualization

3.1 Preparations

MemorySink holds the most recent events in a ring buffer that can be bound to a visualization. UpdateVisu is called cyclically (e.g., in the main program or in the VISU_TASK) to refresh the displayed entries.
To display it, the VisuLoggerList visualization is instantiated in a frame and passed the MemorySink. The frame's Scaling Type must be set to Fixed for correct display. The number of displayed lines can be adjusted depending on screen size (1–50).

```
// Number of visible MemorySink lines
eErr := memSink.Init(nVisuLines := 30);
```



3.2 Memory Buffer

The events buffered in MemorySink are shown via VisuLoggerList as a scrollable list with timestamp, level, and message.

timestamp	severity	message
27.07.2026 13:54:30.004	error	Critical fault in session sess-batch-88 during emergency_stop
27.07.2026 13:54:29.496	error	Sterilization temp 132.8°C exceeds critical max, aborting cycle
27.07.2026 13:54:28.992	warning	Sterilization temp 121.4°C nearing limit 125.0°C
27.07.2026 13:54:28.486	info	Recipe write acknowledged, non critical
27.07.2026 13:54:27.980	info	Operator jjones (badge 4711) acknowledged alarm, credentials Sunshine2024! verified
27.07.2026 13:54:27.474	error	Mixer speed 1150 rpm exceeds critical threshold, controlled stop
27.07.2026 13:54:26.968	warning	Mixer speed 980 rpm approaching limit 1000 rpm

4 Safety Notes

Note: Draft. This section is being created as part of CRA preparation (measures M-19 and M-21) and has not yet been approved. Before publication, the statements regarding delivery state must be checked against the actual code and example projects.

The following notes are addressed to integrators and operators. They describe the intended use of safety-relevant functions as well as the residual risks that remain the integrator's responsibility.

4.1 Delivery State

The logger does not transmit or store any data on its own. Every sink must be explicitly declared, initialized, and added to the logger in the application. It is therefore a deliberate decision by the integrator whether log data leaves the controller, and by what path.

4.2 Network Transmission via SyslogUdpSink

SyslogUdpSink transmits log entries per RFC 5424 over UDP. By design, the protocol is unencrypted and unauthenticated; there is neither confidentiality nor integrity protection, and the receiver cannot verify the origin of a message. Log data may allow conclusions about process parameters, operating states, and plant structure.
Intended use: transmission within a trusted, segmented network in which sender and receiver are under the same administrative control.

Note: SyslogUdpSink is not intended for use across network segment boundaries, over radio links, or over public networks. If there are elevated requirements for confidentiality or integrity, a local syslog relay should be provided within the protected segment to forward messages in encrypted form (e.g., syslog over TLS). AS_Logger itself does not provide an encrypted sink.

4.3 Content of Log Entries

The logger does not evaluate the content passed to it. What data is logged is decided by the application. In particular, the following should not be logged: credentials and key material, personal data beyond what is required for operation, and process know-how worth protecting, such as complete recipes.
For cases where sensitive keys in the payload are unavoidable, the RedactionDecorator is available. It can be applied sink-locally, so that values appear in plain text in the visualization, for example, while being masked on a network sink. Masking only applies to the keys registered via AddKey — the list must be kept up to date as the application changes.

4.4 File Output via FileSink

FileSink writes log files to the controller's file system. Protection of these files against unauthorized reading depends on the controller's access rights and is the operator's responsibility. When choosing file size and archive count, consider how long log data remains on the device and who can access the device.

4.5 Reporting Security Issues

We welcome reports of potential security vulnerabilities in this library and handle them according to our coordinated disclosure policy. (Contact address and link to the policy will be added once the associated measures have been completed.)

5 License

For licensing information about this library, please contact ASKS GmbH at support@as-ks.de or via our website www.as-ks.de.