Document Title	Specification of Operating System Interface
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	719

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	19-03

Document Change History

Date	Release	Changed by	Description
2019-03-29	19-03	AUTOSAR Release Management	• Clarified that PSE51 following POSIX-1003.1-2003 is the currently-targeted version. • Minor changes in tracing, clean up
2018-10-31	18-10	AUTOSAR Release Management	• Add Resource Control • Added Shared object support
2018-03-29	18-03	AUTOSAR Release Management	• Minor changes
2017-10-27	17-10	AUTOSAR Release Management	• Minor changes, document clean up
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction and Functional Overview 4

2 Acronyms and Abbreviations 5

3 Related Documentation 6

3.1 Input Documents & Related Standards and Norms 6
3.2 Related specification 6

4 Constraints and assumptions 7

4.1 Limitations 7
4.2 Applicability To Car Domains 7

5 Dependencies to other modules 8

6 Requirements Tracing 9

7 Functional specification 11

7.1 Operating System Specification 11
7.1.1 Operating System Overview 11
7.1.2 Process Handling 11
7.1.3 Scheduling Policies 13
7.1.4 Time Triggered Execution 14
7.1.5 Device Support 14
7.1.6 Resource control 14
7.2 API Specification 16
7.2.1 Application Interface C (POSIX PSE51) 16
7.2.2 Application Interface C++11 17

A Not applicable requirements 18

1 Introduction and Functional Overview

This document is the software specification of the Operating System Interface within the AUTOSAR Adaptive Platform.

AUTOSAR Adaptive Platform does not specify a new Operating System for highly performant microcontrollers. Rather, it defines an execution context and programming interface for use by Adaptive Applications.

Note that this Operating System Interface (OSI) specification contains application interfaces that are part of ARA, the standard application interface of Adaptive Application. The OS itself may very well provide other interfaces, such as creating processes, that are required by Execution Management to start an Application. However, the interfaces providing such functionality, among others, are not available as part of ARA and it is defined to be platform implementation dependent.

The OSI provides both C and C++ interfaces. In case of a C program, the application's main source code business logic include C function calls defined in the POSIX standard, namely PSE51 defined in IEEE1003.13 [1]. During compilation, the compiler determines which C library from the platform's operating system provides these C functions and the application's Executable must be linked against at runtime. In case of a C++ program, application software component's source code includes function calls defined in the C++ Standard and its Standard C++ Library.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the Operating System Interface that are not included in the [2] AUTOSAR Glossary.

Abbreviation / Acronym:	Description:
OSI	Operating System Interface
Operating System Interface	A Functional Cluster within the Adaptive Platform Foundation
AUTOSAR Adaptive Platform	see [2] AUTOSAR Glossary
Adaptive Platform Foundation	see [2] AUTOSAR Glossary
Adaptive Application	see [2] AUTOSAR Glossary
Execution Management	A Functional Cluster within the Adaptive Platform Foundation
Application	see [2] AUTOSAR Glossary
Operating System	Software responsible for managing Processes on a Machine and for providing an interface to hardware resources.
Process	see [2] AUTOSAR Glossary
Initial Process	A process with management rights, e.g. to determine exit status, for all processes within the AUTOSAR Adaptive Platform.
Foundation	see [2] AUTOSAR Glossary
Machine	see [2] AUTOSAR Glossary
Process	see [2] AUTOSAR Glossary
Executable	see [2] AUTOSAR Glossary
Functional Cluster	see [2] AUTOSAR Glossary

3 Related Documentation

3.1 Input Documents & Related Standards and Norms

[1] IEEE Standard for Information Technology- Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support
https://standards.ieee.org/findstds/standard/1003.13-2003.html

[2] Glossary
AUTOSAR_TR_Glossary

[3] General Specification of Adaptive Platform
AUTOSAR_SWS_General

[4] Requirements on Operating System Interface
AUTOSAR_RS_OperatingSystemInterface

3.2 Related specification

AUTOSAR provides a general specification [3] which is also applicable for `Execution Management`. The specification SWS General shall be considered as additional and required specification for implementation of `Execution Management`.

4 Constraints and assumptions

4.1 Limitations

This chapter lists known limitations of this software specification. The intent is to not only provide a specification of the current state of the `Operating System` interface but also an indication how the `AUTOSAR Adaptive Platform` will evolve in future releases.

The following functionality is mentioned within this document but is not fully specified in this release:

- The currently known limitations are the requirements in [4] which are listed within Appendix A.
- There is currently no sufficient API providing periodic time-based processing to fulfill [RS_OSI_00102]. This will be defined in a future release.

4.2 Applicability To Car Domains

No restrictions to applicability.

5 Dependencies to other modules

There are no dependencies to other AUTOSAR Adaptive Platform standard modules. Any underlying modules required by OS such as bootloader, BSP (Board Support Package), HAL (Hardware Abstraction Layer), BIOS (Basic Input/Output System), and etc., are specific to the OS hence not standardized.

6 Requirements Tracing

The following table references the features specified in [4] and links to the fulfillments of these.

Feature	Description	Satisfied by
[RS_AP_00111]	The AUTOSAR Adaptive Platform shall support source code portability for AUTOSAR Adaptive applications.	[SWS_OSI_01001] [SWS_OSI_01002]
[RS_AP_00113]	API specification shall comply with selected coding guidelines.	[SWS_OSI_NA]
[RS_AP_00114]	C++ interface shall be compatible with C++11.	[SWS_OSI_01002]
[RS_AP_00115]	Namespaces.	[SWS_OSI_NA]
[RS_AP_00116]	Header file name.	[SWS_OSI_NA]
[RS_AP_00119]	Return values / application errors.	[SWS_OSI_NA]
[RS_AP_00120]	Method and Function names.	[SWS_OSI_NA]
[RS_AP_00121]	Parameter names.	[SWS_OSI_NA]
[RS_AP_00122]	Type names.	[SWS_OSI_NA]
[RS_AP_00124]	Variable names.	[SWS_OSI_NA]
[RS_AP_00125]	Enumerator and constant names.	[SWS_OSI_NA]
[RS_AP_00127]	Usage of ara::core types.	[SWS_OSI_NA]
[RS_AP_00128]	Use of exceptions in API.	[SWS_OSI_NA]
[RS_AP_00129]	Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation.	[SWS_OSI_NA]
[RS_AP_00130]	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment.	[SWS_OSI_NA]
[RS_AP_00131]	Use of verbal forms to express requirement levels.	[SWS_OSI_01001] [SWS_OSI_01002] [SWS_OSI_01003] [SWS_OSI_01006] [SWS_OSI_01008] [SWS_OSI_01009] [SWS_OSI_01010] [SWS_OSI_01012] [SWS_OSI_01013] [SWS_OSI_01014] [SWS_OSI_01040] [SWS_OSI_02000] [SWS_OSI_02001] [SWS_OSI_02002]
[RS_AP_00132]	Usage of noexcept keyword.	[SWS_OSI_NA]
[RS_AP_00134]	Library destructors shall be tagged with noexcept.	[SWS_OSI_NA]
[RS_OSI_00100]	The Operating System Interface provided to processes shall provide a PSE51-compliant API.	[SWS_OSI_01001] [SWS_OSI_01002] [SWS_OSI_01003] [SWS_OSI_01006]
[RS_OSI_00102]	The Operating System Interface shall support time-triggered execution.	[SWS_OSI_NA]
[RS_OSI_00103]	The Operating System Interface shall support C++.	[SWS_OSI_01002]
[RS_OSI_00104]	The Operating System Interface shall support the reaction on process-external stimuli from devices.	[SWS_OSI_01001]

[RS_OSI_00105]	The Operating System Interface shall support the start of `Execution Management` .	[SWS_OSI_01040]
[RS_OSI_00201]	The Operating System shall provide mechanisms for system memory budgeting.	[SWS_OSI_02000] [SWS_OSI_02001]
[RS_OSI_00202]	The Operating System shall provide mechanisms for CPU time budgeting.	[SWS_OSI_02000] [SWS_OSI_02002]
[RS_OSI_00203]	The Operating System should provide mechanisms for binding processes to CPU cores.	[SWS_OSI_01006] [SWS_OSI_01012]
[RS_OSI_00204]	The Operating System shall support authorized operating system object access for the software entities which are allowed to do so.	[SWS_OSI_NA]
[RS_OSI_00206]	The Operating System shall provide multi-process support for isolation of applications.	[SWS_OSI_01006] [SWS_OSI_01008] [SWS_OSI_01009] [SWS_OSI_01010] [SWS_OSI_01013] [SWS_OSI_01014]
[RS_OSI_00207]	The `Operating System` shall provide the capability to share code and data in an implicit manner.	[SWS_OSI_01013]
[RS_OSI_00208]	The `Operating System` shall only allow processes to access required functionality.	[SWS_OSI_NA]
[RS_SEC_05006]	No description	[SWS_OSI_02001]

7 Functional specification

7.1 Operating System Specification

7.1.1 Operating System Overview

The real-time Operating System in an embedded automotive ECU offers the foundation for dynamic behavior of the software applications. It manages the scheduling of processes and events, the data exchange and synchronization between different processes and provides features for monitoring and error handling. This chapter describes requirements addressed to the Operating System. Applications, in particular Adaptive Applications may not have the system rights to fully use or configure these aspects directly.

7.1.2 Process Handling

[SWS_OSI_01040]{DRAFT} **Start Execution Management as Initial Process.** ⌈ The Operating System shall allow starting the Execution Management as the Initial Process of the AUTOSAR Adaptive Platform. ⌋*(RS_AP_00131, RS_OSI_00105)*

[SWS_OSI_01006]{DRAFT} **Multi-Threading Support** ⌈ The Operating System shall allow running multiple execution contexts (threads) such that the process can execute multiple code flows. ⌋*(RS_AP_00131, RS_OSI_00100, RS_OSI_00203, RS_OSI_00206)*

On multi-core platforms, multiple threads permitted by [SWS_OSI_01006] may execute concurrently on different cores. All the threads belong to some process, so it is possible that multiple threads in the same process may execute on multiple cores concurrently. Additionally, Execution Management requires the ability to bind a specific Process to a core as part of resource management [SWS_EM_02104].

[SWS_OSI_01012]{DRAFT} **Specification of Core Affinity** ⌈ The Operating System shall provide mechanisms for binding processes to CPU cores. ⌋*(RS_AP_00131, RS_OSI_00203)*

In general, a process provides at least the following:

- A `main()` function as the entry point of the first execution thread of the process.
- A local memory context (address space), providing local, non-shared memory, that includes at least the code, data and heap of the process.
- Some level of memory protection, such that incorrect or invalid memory accesses are detected by the underlying Operating System.
- Operating System descriptors permitting access to OS managed resources.

[SWS_OSI_01008]{DRAFT} **Multi-Process Support** ⌈ The Operating System shall support multiple processes. ⌋*(RS_AP_00131, RS_OSI_00206)*

[SWS_OSI_01009]{DRAFT} **Multi-Process Isolation** ⌈ The Operating System shall isolate each process from one another such that an incorrect or invalid memory access is detected by the Operating System. ⌋*(RS_AP_00131, RS_OSI_00206)*

[SWS_OSI_01014]{DRAFT} **Multi-Process Creation Capability Restriction** ⌈ The Operating System shall allow configuring a process to be forbidden from creating other processes. ⌋*(RS_AP_00131, RS_OSI_00206)*

[SWS_OSI_01010]{DRAFT} **Virtual Memory** ⌈ Operating System shall execute each process in a dedicated address space. ⌋*(RS_AP_00131, RS_OSI_00206)*

Each process has its own logical address space where the code and data are located. The address may or may not correspond to their underlying physical address space as the process's address space is virtualized. In particular, multiple instances of the same Executable running in different logical address spaces may share the physical address for its code and read-only data, as they are read-only, to save some physical memory. The rewritable data, on the other hand, need to be separate, so they are mapped to different physical addresses.

Shared objects (also sometimes called DLLs) usually consist of code and data usable from multiple processes simultaneously. When multiple processes use a shared object, code and read-only data is usually mapped in each process but present only once in system memory, while shared data may be duplicated immediately or when needed (Copy-on-Write, or CoW).

Shared objects can be used in mainly two ways:

- *Implicit loading*: at build time, an Executable may be linked against a shared object ; later on, at load time, the Operating System and its loading framework enable the mapping and use of the shared object code and data in the process of the application. This is mainly used for space saving and ease of deploying fixes in shared code, but sometimes also for licensing reasons. The process itself does not require any specific capability or knowledge of this shared library existence to make use of it.
- *Explicit loading*: at run time, the Process requests the Operating System and its loading framework to open and load a shared object on the target, and to let it resolve symbol names and load its code and data. This is usually done for plugin mechanisms where all plugins expose the same shared symbols. The Executable itself has no knowledge of the plugins at link time, and typically uses the `dlopen()`/`dlsym()`/`dlclose()` to enable using the plugin-style loaded shared object.

[SWS_OSI_01013]{DRAFT} **Implicit shared object support** ⌈ The Operating System shall allow the use of Implicit loading of shared objects for Executables. ⌋*(RS_AP_00131, RS_OSI_00207, RS_OSI_00206)*

Note that for safety, security or other reasons, an Executable may be built fully statically-linked, and therefore not use the capability to use shared objects.

7.1.3 Scheduling Policies

The Operating System Scheduler is designed to keep all system resources busy allowing multiple software control flows to share the CPU cores in an effective manner. The main goals of the scheduling mechanisms may be one or more from the following:

- Maximizing throughput in terms of amount of work done per time unit.
- Maximizing responsiveness by minimizing the time between job activation and actual begin of data processing.
- Maximizing fairness in terms of ensuring appropriate CPU time according with priority and workload of each job.
- Assuring a timelined and ordered activation of jobs according to some policy-dependent job execution eligibility (e.g. priority, deadline, tardiness, etc).

In real life these goals are often in conflict, implementing the scheduling mechanisms is therefore always a compromise.

[SWS_OSI_01003]{DRAFT} **Default Scheduling Policies** ⌈ The AUTOSAR Adaptive Platform Operating System shall support the following scheduling policies defined in the IEEE1003.1 POSIX standard: SCHED_OTHER, SCHED_FIFO, SCHED_RR. ⌋*(RS_AP_00131, RS_OSI_00100)*

In order to overcome the above mentioned conflicts and to achieve portability between different platforms, the AUTOSAR Adaptive Platform Operating System provides the following scheduling policies categorized in two groups:

- Fair Scheduling Policies
  - SCHED_OTHER
- Real-time Scheduling Policies
  - SCHED_FIFO
  - SCHED_RR

Since the above mentioned default scheduling policies may not guarantee proper execution for all real-time scenarios, the Adaptive Application vendor may provide additional scheduling policies to fulfill any execution requirement. For example, additional non-POSIX scheduling policies like SCHED_DEADLINE (Earliest Deadline First algorithm) could be introduced to satisfy hard real-time requirements.

7.1.4 Time Triggered Execution

POSIX PSE51 provides a means to do time-based periodic processing, using the timer API (e.g. `timer_settime()`) along with POSIX signals. However, signals are sometimes discouraged for safety-critical applications, because they disrupt the execution flow.

Using C++, `std::future::wait_until()` can be used to realize periodic processing. The TimeSync specification may also be used along with std::future to provide event generation. However, both of these APIs only allow single-shot, relative alarms, and efficient, low-overhead requires recurring and/or absolute alarms.

Therefore, these APIs may be extended in the future.

7.1.5 Device Support

The OSI shall support device access as defined in POSIX PSE51.

7.1.6 Resource control

While correct behavior is expected from each application, intentional or unintentional misbehavior must be contained for system stability. Simultaneously, some level of dynamic behavior must be allowed. From a feature perspective, applications can be assembled in groups such that they can follow a similar usage pattern, sharing memory, CPU time, and in general resources.

[SWS_OSI_02000]{DRAFT} **Resource Group minimum requirement** ⌈ The `Operating System` shall support the configuration of at least 8 groups of processes in the system. ⌋*(RS_AP_00131, RS_OSI_00201, RS_OSI_00202)*

Depending on the `Operating System`, the number of usable Resource Groups may vary. Furthermore, when OS-level-virtualized containers are used, some `Operating Systems` may additionally constrain the number of usable Resource Groups, with an extreme of just 1 available Resource Group.

[SWS_OSI_02001]{DRAFT} **Memory Resource Groups** ⌈ The `Operating System` shall support a mechanism to define groups of processes that may dynamically allocate memory from a configuration-defined limit. ⌋*(RS_AP_00131, RS_OSI_00201, RS_SEC_05006)*

The memory taken in consideration for the limit covers:

- Code and read-only Data from the `Executable`
- Modifiable Data from the `Executable`
- Memory used for thread stack for each thread of the process

- Heap
- System memory that is used by the Operating System for holding the kernel resources allocated to the process (e.g. thread control block, semaphore, page table entries for MMU mapping, etc)
- Shared memory between processes of the same group
- Implicitly loaded shared objects between processes of the same group

Because memory accounting may differ between Operating Systems, some elements can be considered inside or outside the memory usage limit of the process group, in an implementation-specific manner:

- Shared memory between processes of different groups
- Memory-mapped files
- Implicitly loaded shared objects between processes of different groups

[SWS_OSI_02002]{DRAFT} **CPU Resource Groups** ⌈ The Operating System shall support a mechanism to define groups of processes that may use a maximum configured amount of CPU time over a defined period of time. ⌋*(RS_AP_00131, RS_OSI_00202)*

Because scheduling is done in very different ways depending on the Operating System, the specific algorithm for scheduling as well as limiting the CPU usage is not described here.

Example valid group scheduling schemes include (but not limited to):

- Fixed-periodic enablement of processes over a fixed range of time, in a manner similar to what the ARINC 653 standard defines.
- Processes use time from a quota of time allocated to the group. If no time remains, no thread from the processes in the expired group can be scheduled. Each period, the quota is replenished to allow more time to be used and corresponding threads to be scheduled again.
- Processes accumulate time usage. Each period or each context switch, time usage accumulated over a certain count of past periods is calculated. Processes of each group that used time over a threshold are disabled, and processes of each group that used time under a threshold are enabled.

Most notably, on some Operating Systems, idle time, which by definition is not requested to be used by any process group, may be distributed to any process, including those belonging to a group that is considered to be using time over the defined limit. This is a worthy optimization, but is currently not considered in the specification as a requirement.

7.2 API Specification

The AUTOSAR Adaptive Platform does not specify a new Operating System for highly performant microcontrollers. Rather, it defines an execution context and programming interface for use by Adaptive Applications.

7.2.1 Application Interface C (POSIX PSE51)

[SWS_OSI_01001]{DRAFT} **POSIX PSE51 Interface** ⌈ The OSI shall provide OS functionality with POSIX PSE51 interface, according to the 1003.13-2003 specification. ⌋*(RS_AP_00131, RS_OSI_00100, RS_OSI_00104, RS_AP_00111)*

Note that PSE51 requires C99 as specified in the standard.

There are several Operating Systems on the market, e.g. Linux, that provide POSIX compliant interfaces. However Applications are required to use a more restricted API to the Operating Systems as compared to the platform services and foundation. In particular, the starting assumption is that an Adaptive Application may use PSE51 as OS interface whereas platform-specific Application may use full POSIX.

The implementation of platform Foundation and platform services functionality may use non-PSE51 APIs, even OS specific ones. The use of specific APIs will be left open to the implementer of the AUTOSAR Adaptive Platform and is not standardized.

In case of a C program, the applications main source code business logic includes C function calls defined in the POSIX standard. During compilation, the compiler determines which C library from the platforms Operating System provides these C functions and the applications executable must be linked against at runtime. This Operating System provided C library can implement the POSIX-compliant C function in two ways:

- The provided C library implements the behavior as part of the library. Then, the execution of this C function causes no further invocation of the Operating System with a system call.
- The provided C library implements the behavior through a suitable system call of the Operating System kernel. In many cases, the function name and behavior of the Operating System kernel system call match very closely to the Operating System provided C library and to the POSIX-specified function definitions. For example, in the case of typical Linux distributions, these functions are provided by `glibc` library, and by default, the `gcc` compiler links the `glibc` library dynamically.

7.2.2 Application Interface C++11

[SWS_OSI_01002]{DRAFT} **Use of C++ Language** ⌈ The OSI shall provide OS functionality with C++11 Standard Library for `Applications` written in C++. ⌋ *(RS_AP_00131, RS_OSI_00100, RS_OSI_00103, RS_AP_00111, RS_AP_00114)*

In case of a C++ program, application software components source code can include function calls defined in the C++11 Standard and its Standard C++ Library. The C++ Standards defines C++ Standard Library (http://en.cppreference.com/w/cpp), and it includes Thread support library, Input/output library and others that provide most of PSE51 functionalites through these C++ interfaces. Some PSE51 functions, such as setting thread scheduling policies, are not available yet through these C++ Standard Library and C++ applications need to use PSE51 C interface in conjunction with these C++ libraries.

In case of Linux and the `gcc` C++ compiler (`g++`), the compiler links the `libstdc++` library, which provides the defined Standard C++ library functions. The `libstdc++` library itself depends on the `glibc` library, i.e., the `libstdc++` implementation includes function calls to the glibc library.

A Not applicable requirements

[SWS_OSI_NA]{DRAFT} ⌈ These requirements are not applicable as they are not within the scope of this release. ⌋*(RS_OSI_00102, RS_OSI_00204, RS_AP_00113, RS_AP_00115, RS_AP_00116, RS_AP_00119, RS_AP_00120, RS_AP_00121, RS_AP_00122, RS_AP_00124, RS_AP_00125, RS_AP_00130, RS_AP_00127, RS_AP_00128, RS_AP_00129, RS_OSI_00208, RS_AP_00132, RS_AP_00134)*