Document Title	Demonstrator Design of Functional Cluster Persistency
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	859

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	• Introduced ara::per::ifc::Initialize and ara::per::ifc::Deinitialize • Introduced mechanisms for handling CppImplementationDataTypes • Introduced storage location identification through ara::core::InstanceSpecifier
2019-11-28	R19-11	AUTOSAR Release Management	• Removed information about header files, classes, methods, functions available via Doxygen • Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Classes and functions of FS and KVS renamed • Introduced ara::core types in all API functions • Introduced handlers to access KVS and FS • Recovery of KVS based on redundant data

2018-11-02	18-10	AUTOSAR Release Management	• Introduced exception-less API (ara::core::Result) • Improved internal structure and test coverage • Harmonized access to File-Proxy and KVS database • Removed WriteAccessor class
2018-05-02	18-03	AUTOSAR Release Management	• Initial support for AUTOSAR data types • Removed KvsType from public API • Initial support for redundancy • Improved access to logging and CRC
2017-11-22	17-10	AUTOSAR Release Management	• Rework of FileProxy/Stream API • Minor changes and bugfixes
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

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

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply to 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 Persistency demonstrator code is to provide a proof of concept for Persistency features specified in AUTOSAR Adaptive.

1.1 Known limitations

This chapter lists all known limitations, not yet implemented features and missing SWS items for the FC Persistency.

- Binary data according to SWS_PER_00303 is supported in the form of byte vectors or strings.
- The UCM use cases are not yet implemented.
- Privacy of data not enforced by implementation.
- Encryption/decryption not supported.
- The implementation of statistics related methods doesn't take into account the storage used by redundancy (CRC and M out of N).
- Reset to initial value functionality is missing for FileProxy/KeyValueStorage
- The classes KeyValueStorage and FileStorage are currently implemented as abstract classes.

2 Overview on architecture

2.1 Design approach / Design principles

The cluster Persistency offers two mechanisms to access non-volatile memory on an Adaptive machine: File Storage and Key-Value Storage.

Persistency is implemented as a library which enables it to be run in the context of an application. This eliminates the need to explicitly manage rights and roles that were configured for the application. Also, there is no concurrent resource access by several processes as resources are owned exclusively by one process. Finally, this approach enables application developers to implement a central persistency platform module or -

for example - a dedicated persistency wrapper application with which other applications can communicate, for example using ara::com.

In order to start and shut down all functional clusters which implements direct ARA interfaces, the application should use ara::core::Initialize and ara::core::Deinitialize. To ensure a correct initialization/de-initialization procedure, Persistency should implement its own pair of ara::per::ifc::Initialize and ara::per::ifc::Deinitialize, which will be triggered by ara::core::Initialize and ara::core::Deinitialize.

Persistency functional cluster is able to take care of the integrity of the stored data. The Persistency cluster can use redundant information to detect data corruption. In this respect, there are two types of measures supported by Persistency cluster: CRC and "M out of N" schema. Those are configurable during deployment and can be used indipendantly or together.

Persistency functional cluster is able to generate statistics regarding the amount of used resources by calling: GetCurrentKeyValueStorageSize, GetCurrentFileStorage Size or GetCurrentFileSize.

Persistency cluster is prepared to handle concurrent access from multiple threads of the same application, running in the context of the same Process. To create shared access to a Key-Value Storage or File Storage, either the SharedHandle returned by OpenKeyValueStorage and OpenFileStorage can be passed on (i.e. copied) to another thread, or OpenKeyValueStorage and OpenFileStorage can be called in independent threads for the same Key-Value Storage or File Storage, respectively.

2.1.1 Key-Value Storage

Key-Value Storage is implemented by providing a class KeyValueStorage. Every instance of this class represents one Key-Value database, so an Application can use multiple instances of this class depending on the number of databases used.

Creating an instance of this class with OpenKeyValueStorage will load all Key-Value elements of the database into RAM.

Afterwards these elements can be accessed by using the APIs GetValue and SetValue.

The synchronization to the physical storage medium will be triggered by calling the method SyncToStorage.

In order to remove all pending changes since the last call of SyncToStorage or since the KeyValueStorage was opened using OpenKeyValueStorage, the method Discard PendingChanges should be called.

Persistency is able to store any CppImplementationDataTypes which are properly described in ARXMLs configuration files. In order to access these elements the GetValue should be called as in the following snippet:

```
auto db = ara::per::OpenKeyValueStorage(kvsInstanceSpecifier).ValueOrThrow();
auto value = db->GetValue<cppImplementationDatatype>(keyName).ValueOrThrow();
db->SetValue(keyName, newValue).ValueOrThrow();
```

2.1.2 File Storage

File access is implemented by providing the classes ReadAccessor and ReadWrite Accessor.

The File Storage is accessed using the function OpenFileStorage.

The single files inside of a File Storage are accessed using the methods OpenFileRead Only, OpenFileWriteOnly and OpenFileReadWrite provided by the class FileStorage.

2.2 Dependencies on other Functional Clusters

For debugging purposes the ara::log library is utilized. The CRC calculation in the Json Parser class is achieved by using the libapd_crc.

2.3 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
RapidJSON	1.1.0	Database format for the key-value storage-	header only, static linked	MIT	NA
Google Test	1.7.0	Unit testing	Not relevant	3-Clause BSD	NA

2.4 Class Overview

2.4.1 KeyValueStorage

The class KeyValueStorage offers all methods to access a Key-Value storage and its elements and is created using OpenKeyValueStorage and passed to the application within a SharedHandle.

2.4.2 KvsType

KvsType is only used internally to keep key-value pairs in RAM.

The class KvsType is a "smart union", which provides support for storing the basic types (POD), extented with std::string, arrays of nested objects and binary-format support, which stores mem-copyable content.

The class provides a collection of constructors for saving the requested type, with internal meta-data, such as the stored type checksum, and of course the relevant key. The stored values can be cast back using the provided GetXXX-APIs.

Additionally, the API provides methods for checking the validity of the data.

The actual implementation is hidden by using the pImpl-pattern.

Usage examples can be found from keyvaluestorage test cases.

2.4.3 FileStorage

The class FileStorage offers methods to access a File Storage and its files and is created using OpenFileStorage and passed to the application within a SharedHandle.

It uses the two classes ReadAccessor and ReadWriteAccessor to provide access to its files.

The Persistency cluster offers the user different classes for accessing data while fulfilling the PSE51-requirements, which most of all, denies creation and deletion of new files. This is guaranteed by definition of a PortPrototype typed by a PersistencyFile ProxyInterface which is a hint to a Platform integrator that there needs to be a chunk of memory (e.g. folder, address space or one big file) reserved in which the application implementation can create multiple files/accessors and interact with them using methods like getline, write or the stream operators.

2.4.4 ReadAccessor

The class ReadAccessor offers methods to read the content of a file and is created using OpenFileReadOnly or OpenFileReadWrite and passed to the application within a UniqueHandle.

2.4.5 ReadWriteAccessor

The class ReadWriteAccessor offers methods to read/write the content of a file and is created using OpenFileReadWrite or OpenFileWriteOnly and passed to the application within a UniqueHandle.

It inherits from ReadAccessor.

2.4.6 SharedHandle

The class SharedHandle is used to provide shared access to a Key-Value Storage or a File Storage.

2.4.7 UniqueHandle

The class UniqueHandle is used to provide non-shared access to a file.

3 Implementation Details

3.1 Database format

The database is currently implemented using Json, but the public API leaves the implementation open, it could as well be XML or anything else.

The used Json schema is defined here: ara-api/src/ara/per/key-value-storage/config/json-schema/kvs-json-schema.json

The schema defines the supported keys, supported types, and the ranges of the types.

A simple example database:

```
[
    {
        "key": "int8array",
        "value": {
            "sint8[]": [2,4,-55]
        },
        "checksum": 3612473348
    },
    {
        "key": "hello-key-string",
        "value": {
            "string": "hello-adaptive!"
        },
        "checksum": 2525760749,
        "documentation":"some documentation about this element"
    }
]
```

- Key : This is the key used for searching the stored value
- Value: This is the actual stored value. (A json object)
  - This value can be any of the supported types, eg. uint8,16,32, ..., string, object)
  - Arrays are marked with []

  - Object is a special value, which denotes that the value stored is actually an
  - object, which may contain more object, object arrays, or plain types.
  - This allows arbitrary deep tree structures.

- Checksum: In this implementation, this is a crc32 calculated over the KVSElement
- Documentation: The schema allows storing some documentation about the stored value. This is intended for manual saving of meta-data, which is not used in any way by the implementation, nor there is no API to read or write this data.