Mentor® Automotive

User Guide
Key-value-store

Version 2.3

<16/01/2016>

Content

List of Tables & Figures

Tables

Figures

1. Introduction

1.1 Purpose

The scope of this document covers detailed interface description, building the library, testing the component and debugging. The key-value store backend for the Persistence Common Object will be described here which has been implemented by Mentor Graphics.

1.2 Revision History

Revision	Change	Date
V 1.0	Setup of Document	30.04.2014
V 1.1	Updated known issues section	20.05.2014
V 1.2	Updated testing and debugging section	11.06.2014
V 1.3	Updated How to build section	06.06.2014
V 1.4	Updated Dynamic Behavior section	01.08.2014
V 1.5	Updated How to build and testing section	10.08.2014
V 1.6	Updated Dynamic Behavior section	15.08.2014
V 1.7	Added file structure Backup / Rec. section	22.09.2014
V 1.8	Updated testing and debugging section	11.11.2014
V 1.9	Updated database file structure section	02.12.2014
V 2.0	Updated sections: **Fehler! Verweisquelle konnte nicht gefunden werden.**; REF _Ref413394730 \r \h **Fehler! Verweisquelle konnte nicht gefunden werden.**; **Fehler! Verweisquelle konnte nicht gefunden werden.**; **Fehler! Verweisquelle konnte nicht gefunden werden.**; **Fehler! Verweisquelle konnte nicht gefunden werden.**	05.05.2015
V 2.1	Updated section **Fehler! Verweisquelle konnte nicht gefunden werden.**	10.08.2015
V 2.2	Switched to new template, documentation rework	30.10.2015
V 2.3	Changed document license to CC BY-SA 4.0	16.01.2016

Table 1 - Revision History

1.3 Abbreviations & Terminology

Abbreviation	Description
API	Application Programming Interface
PCL	Persistence Client Library
PCO	Persistence Common Object
RCT	Resource Configuration Table

Table 2 – Abbreviations & Terminology

2. Persistence Common Object

2.1 Persistence Subsystem

The Persistence subsystem is responsible for handling persistent data. Persistent data is data that needs to be stored on a head unit between restarts, it includes all data read and modified during the lifetime of an infotainment system. Persistent data is stored in a non-volatile storage such as a hard disk drive or FLASH memory.

GENIVI defines the structure for the persistence management to meet standardization of GENIVI compliant platforms and some unique requirements introduced by the automotive domain.

The Persistence Management consists of the following components:

- Persistence Client Library
- Persistence Administration Service
- Persistence Common Object
- Persistence Health Monitor

There are a few reasons why GENIVI is developing this area, as opposed to simply saying that "many Linux storage solutions exist" and leaving it at that. First of all the overall goal is standardization of GENIVI compliant platforms so that applications running on GENIVI all use a common service for storing data. Secondly the automotive domain introduces some unique requirements:

- Strong robustness and reliability requirements from automotive OEMs for storing of critical data.
- Unreliable electrical environment, with possible power drop-out (engine cranking, blown fuses etc.)
- Lifetime requirements of automotive vehicle (in the order of 10-20 years) balanced against the lifetime of flash memory with its limited number of write (and to some extent read) cycles

The overall idea is to define a standardized common interface to persistent storage, under which system builders will still have significant flexibility in choosing the actual storage implementation that makes sense for each system. The persistence client library also provides an abstraction that avoids clients having to handle specific paths in the VFS and similar implementation details.

2.1.1 Persistence Components in System Context

The image below shows the persistence components in context with other components.

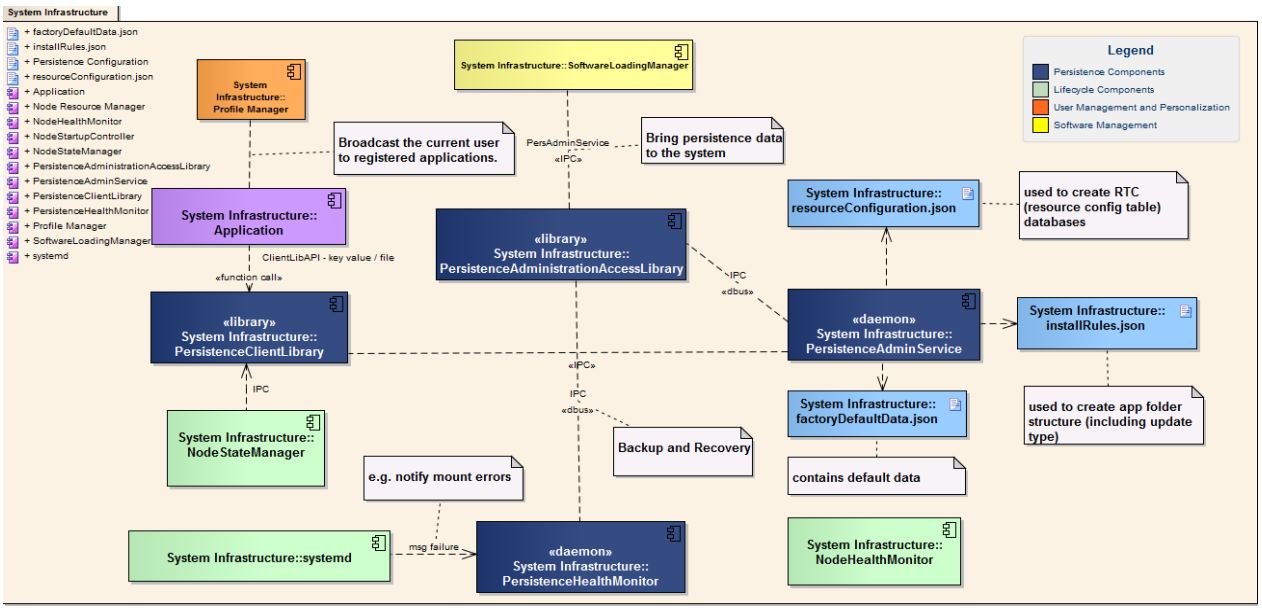

Figure 1 - System Context

2.1.2 Persistence Components Overview

The image below shows all persistence components and their relation.

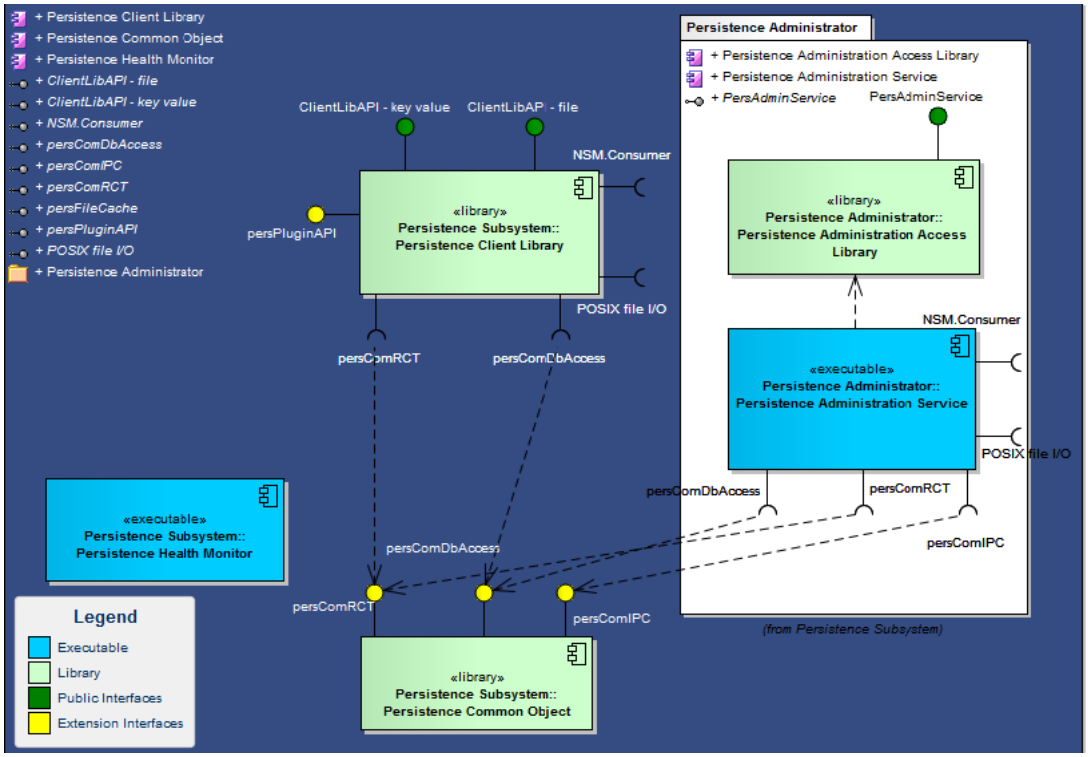

Figure 2 - Persistence Components Overview

2.2 Key-value Store

The Persistence Common Object component is an abstraction of the database and interprocess communication and is provided for the usage in context of the Persistence Client Library (PCL) and the Persistence Administration Service (PAS). The Persistence Common Object Component (PCO) is used by the PCL as a backend to store key/value data and resource configuration table data (RCT). The PCO is also used by the Persistence Administration Service (PAS) to create key/value data and configuration data during software loading process.

The intended usage of the PCO is in combination with the Persistence Client Library (PCL). The PCL uses the PCO to cache the access of key-value pairs via the PCL key-value interface in order to extend the flash memory lifetime.

2.2.1 Known Issues

Currently fixed cache size implemented. This means approx. max. 2100 key-value pairs per database in cache possible if every key value pair uses maximum value size of 8028 byte.

2.3 License

2.4 Cached Key-value Database

The PCO provides a cached database backend which is logically divided into two main parts. One part is the data structure keeping the key-value pairs cached in RAM, and the other part is responsible to store the key-value parts persistently in a file located in a non-volatile storage device.

In order to store the cached data persistently on flash memory, the database backend implements a modified version of the lightweight KISSDB (Keep It) Simple Stupid Database (https://github.com/zerotier/kissdb) implementation which is licensed under public domain.

The caching mechanism on top of the KISSDB implementation stores only key-value pairs in RAM that get modified during the lifecycle to assure optimal memory consumption.

2.5 Caching Strategy

This section describes the caching mechanism in detail.

Reading a key-value pair:

- If an application wants to read a key-value pair which was not modified during this system lifecycle, the data is read from the database in flash memory.
- If the key-value pair to be read was modified or deleted in this system lifecycle, the data is read from the cache in RAM.

Writing a key-value pair:

- If an application inserts a new key-value-pair, the data is inserted into the cache in RAM.
- The inserted data gets stored persistently in flash memory at system shutdown.

Update an existing key-value pair:

- If an application wants to update an existing key-value pair, the data gets updated in the Cache if already present and gets marked as modified. If the data which has to be modified is not already present in RAM it has to be inserted and being marked as "modified". All data modifications are done on the data structure in RAM.
- Data modifications get stored persistently in flash memory at system shutdown.

Deleting an existing key-value pair:

- The key-value pair to be deleted must be marked as deleted on the data structure in RAM. If the key is not already present in this data structure, it must be inserted with no value and be marked as deleted.

2.5.1 Shutdown Scenario

The write back scenario at system shutdown is the following:

- Application receives shutdown notification
- Application writes the data for the last time in this lifecycle
- Application sends notification back to lifecycle that shutdown is OK now
- PCL instance of the application receives shutdown notification
    - PCO instance of the application writes data back from cache to flash memory and closes the database finally, if no other instances have the database opened.

2.6 Database File Structure

The persisted database file on the flash memory includes a header section with the following elements:

- **KDB version:**
    - used to store the version of KISSDB: (current: KDB 2.3)
- **Close failed flag:**
    - flag to indicate if previous close has failed
- **Open ok flag:**
    - flag to indicate if open was successful
- **Hashtable size:**
    - used to store the amount of entries in a hashtable -> i.e 256 byte
- **Key size:**
    - used to store the fixed length of a key in bytes -> i.e. 128 byte
- **Value size:**
    - used to store the maximum length of a value in bytes -> i.e. 8028 byte

The persisted database file on the flash memory includes at least one hashtable section with the following elements:

- **delimiter start:**
  - static value to indicate the beginning of a hashtable section
- **CRC32:**
  - crc value of the hashtable
- **hashtable**
  - slot structure: 510 slots to store the offsets to the datablocks
- **delimiter end:**
  - static value to indicate the end of a hashtable section

Hashtable slot structure:

- **offset A:**
  - absolute offset in bytes to data block A which contains the key / value pair
- **offset B:**
  - absolute offset in bytes to data block B which contains the key / value pair
- **current:**
  - flag to indicate which data block currently contains latest valid data (in current implementation, both datablocks hold latest data)

Key / value pairs are stored in datablocks with the following structure:

- **delimiter start:**
  - static value to indicate the beginning of a datablock
- **checksum:**
  - CRC32 of key and value
- **key:**
  - key with a maximum size of 128 bytes
- **value size:**
  - size of the value to be stored in bytes
- **value:**
  - value to the corresponding key with a maximum size of 8028 byte
- **hashtable number:**
  - number that indicates which hashtable stores the offset for this block
- **delimiter end:**
  - static value to indicate the end of a datablock

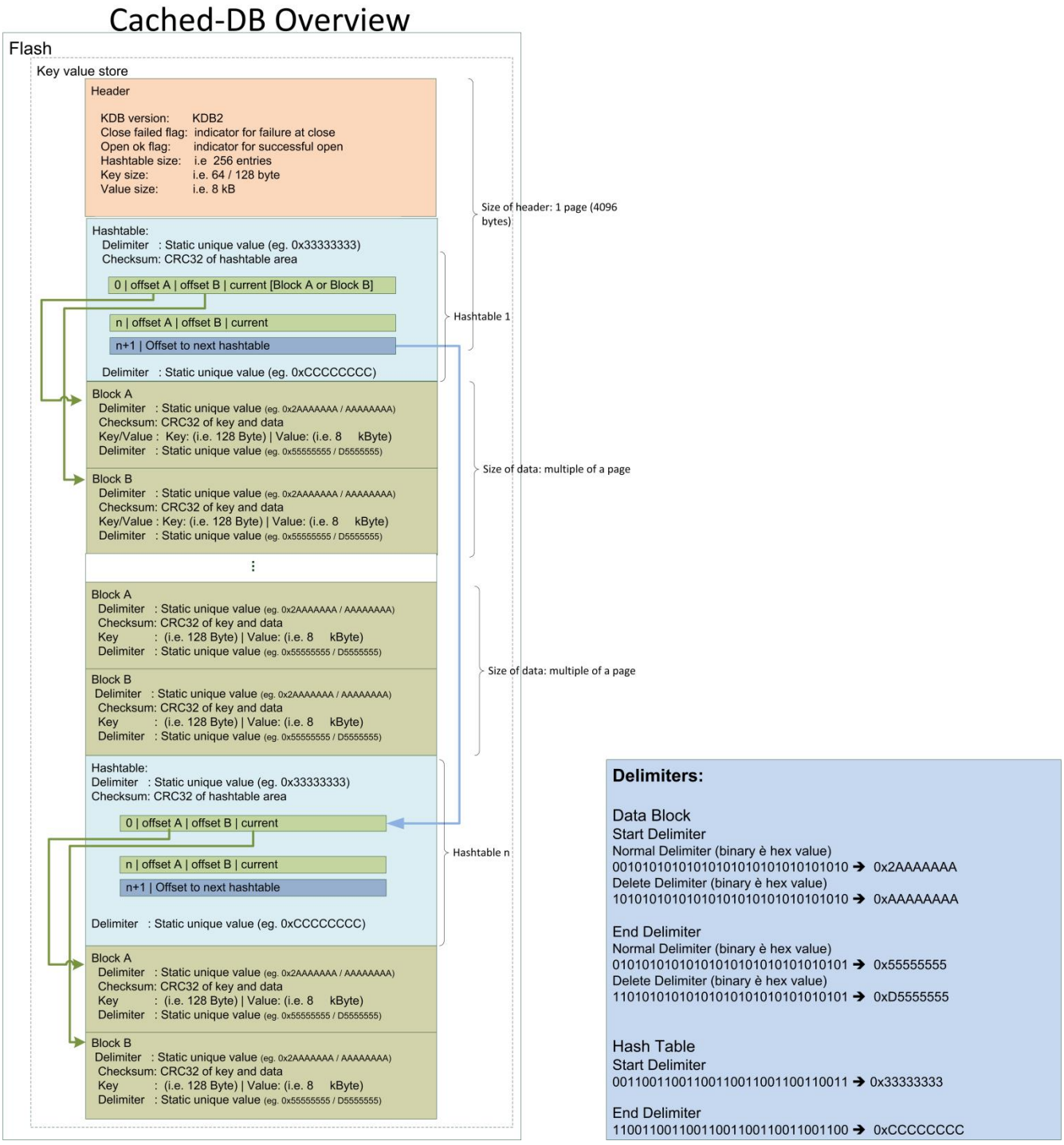

Figure 3 - Database File Details

2.7 Backup and Recovery Mechanism

While opening a database file, a check is made to verify if the database has been closed correctly.

If so, no further actions are taken. If an incorrect close was identified, the checksum of all hashtable sections in the file get verified. If a hashtable checksum is invalid, all hashtables in the database get rebuilt if possible.

If the checksums of all hashtables are valid, a datablock may be corrupted. Therefore a CRC check of all datablocks listed in the hashtables is done.

The next subsections describe in which cases a database recovery is possible.

2.7.1 Database Recovery

If the checksum of a hashtable is invalid, the recovery mechanism tries to rebuild all hashtables in the database file. Datablocks and hashtables in the file are searched on the basis of their delimiters. During this process, the following cases can occur:

- No data or hashtable delimiter was found
- deleted Datablock A start or end delimiter was found
- deleted Datablock B start or end delimiter was found
- Datablock A start or end delimiter was found
- Datablock B start or end delimiter was found
- Hashtable start or end delimiter was found

For all the cases where a datablock was identified (either a datablock start or end delimiter is found), the mechanism checks if the stored data is valid and can be used for recovery. If for example datablock A is corrupt and datablock B is valid, datablock B will be used for recovery.

A full rebuild of all datablocks is not possible for the following scenarios:

- Both delimiters of datablock A and of datablock B are corrupt.
- Both delimiters of a hashtable are corrupt.
- Both delimiters of datablock A are corrupt and data of datablock B is invalid.
- Both delimiters of datablock B are corrupt and data of datablock A is invalid.
- Data of datablock A and of datablock B is corrupt.

For these cases listed above, the persistence client library provides a mechanism to automatically search for available default data in the default databases.

3. Persistence Common Object Library API

The Persistence Common Object Library provides a C-API for the Persistence Client Library (PCL) allowing cached access to key-value databases. The API is not intended to be used directly within other applications.

The API has the following functions:

- `signed int persComDbOpen(char const * dbPathname, unsigned char bForceCreationIfNotPresent)`
- `signed int persComDbClose(signed int handlerDB)`
- `signed int persComDbWriteKey(signed int handlerDB, char const * key, char const * data, signed int dataSize)`
- `signed int persComDbReadKey(signed int handlerDB, char const * key, char* dataBuffer_out, signed int dataBufferSize)`
- `signed int persComDbGetKeySize(signed int handlerDB, char const * key)`
- `signed int persComDbDeleteKey(signed int handlerDB, char const * key)`
- `signed int persComDbGetSizeKeysList(signed int handlerDB)`
- `signed int persComDbGetKeysList(signed int handlerDB, char* listBuffer_out, signed int listBufferSize)`

For more details please refer directly to the header file located /inc/protected/persComDbAccess.h.

4. Dynamic Behavior

4.1 Reading a key-value pair

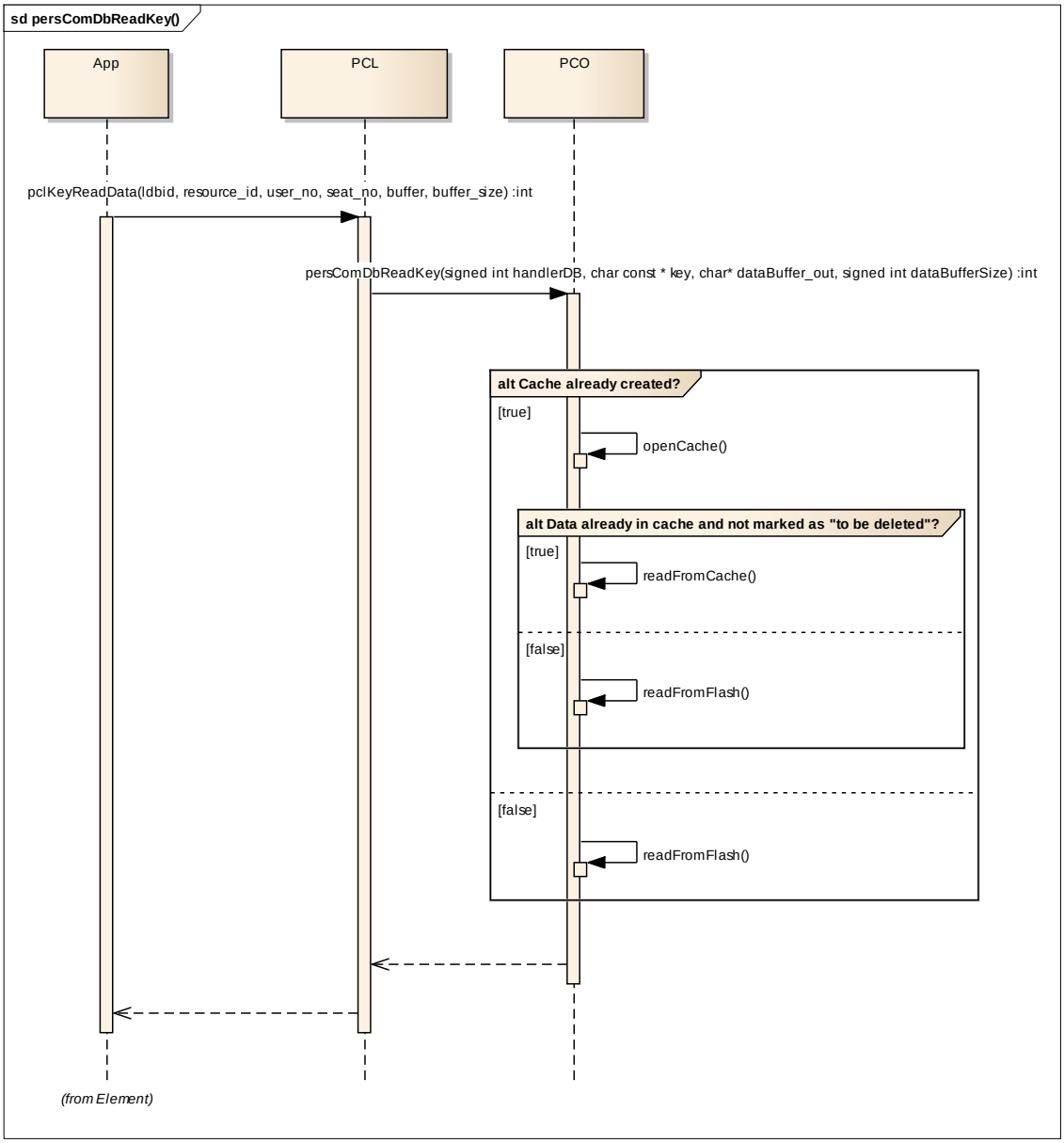

Figure 4 - Reading a key-value pair

4.2 Writing a key-value pair

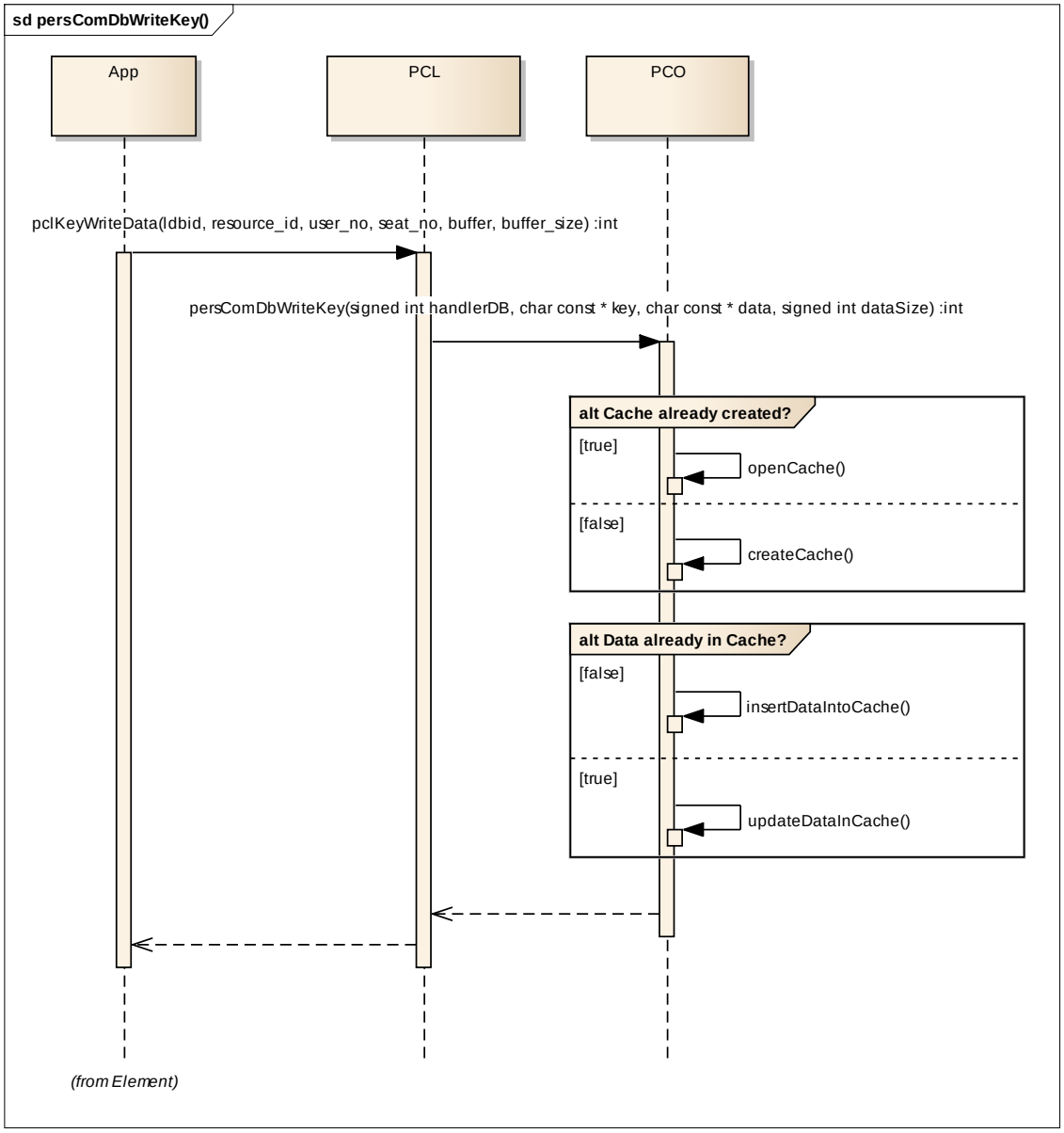

Figure 5 - Writing a key-value pair

4.3 Deleting a key-value pair

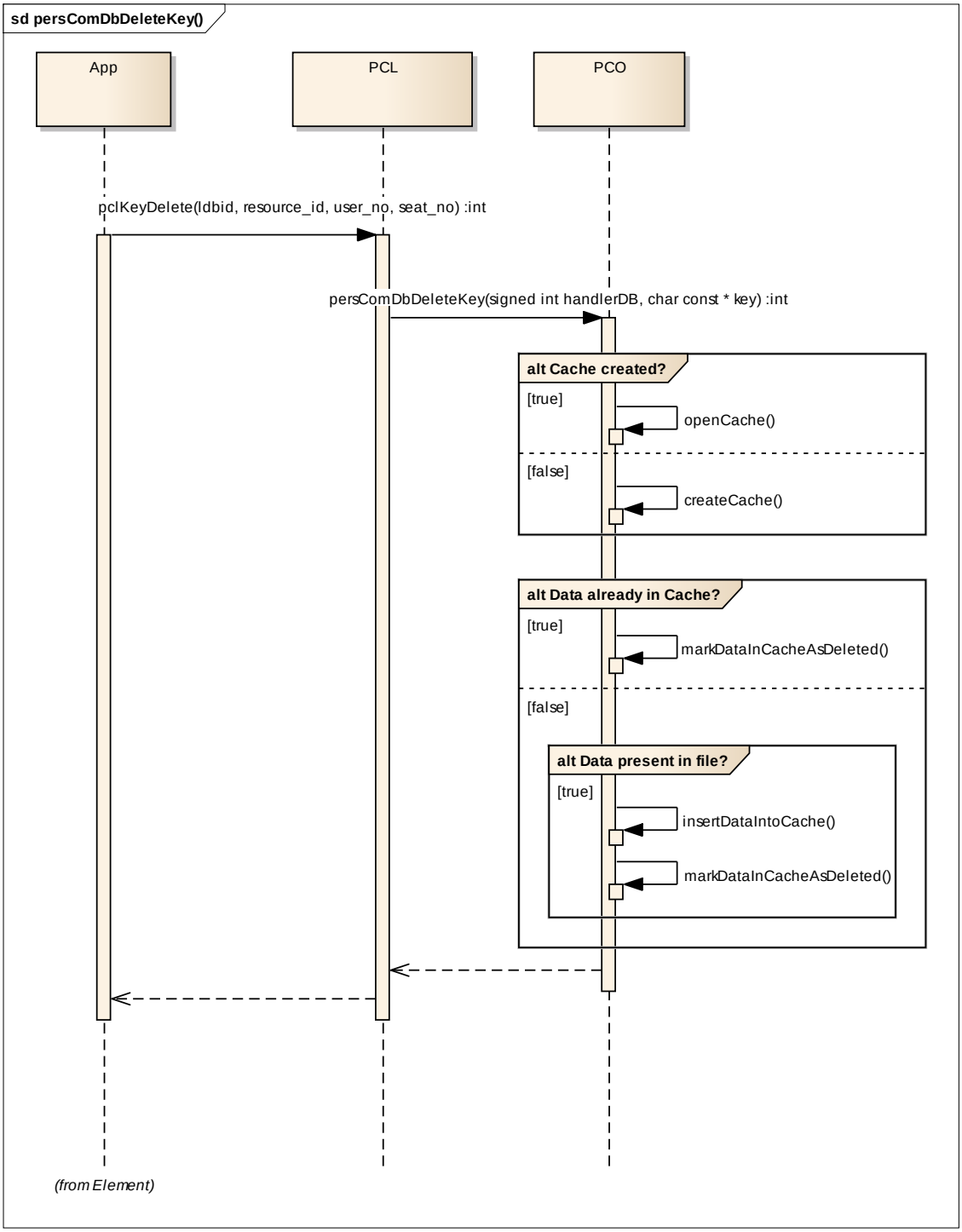

Figure 6 - Deleting a key-value pair

4.4 Persist cached changes to flash

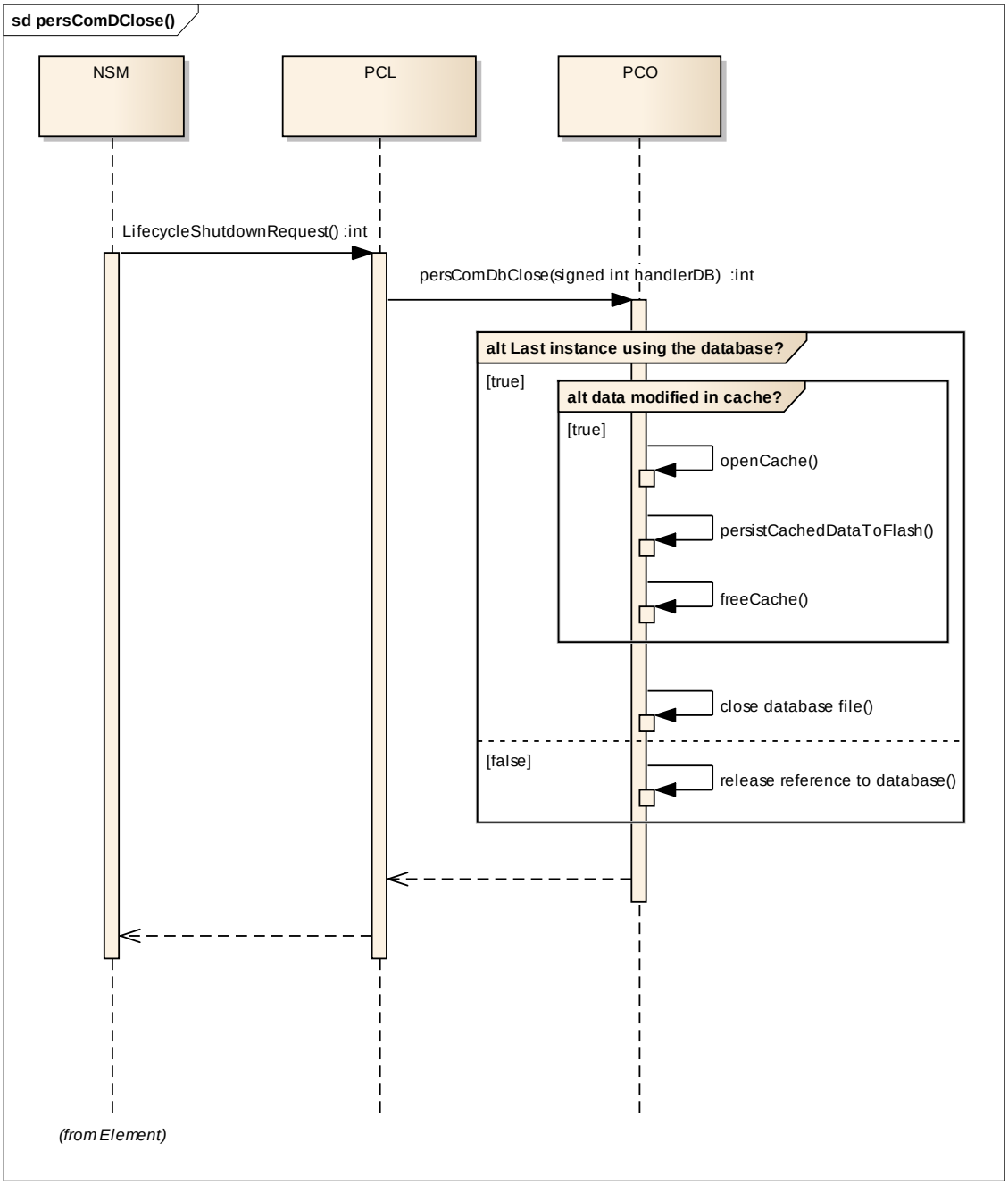

Figure 7 - Persist cache changes to flash

5. How to Build

This chapter provides all the information needed to build the component and run the test cases.

5.1 Dependencies

The client library has the following dependencies

- Components
  - automotive-dlt
    - http://projects.genivi.org/diagnostic-log-trace/
  - check unit test framework for C (
    - used when configured with "--enable-tests"
  - Itzam/C
    - Used when configured with “--with-database= itzam”
    - Warning: Itzam/C is no longer supported, use key-value-store instead
- Tools
  - Autotools
  - libtool

5.2 Building the Library

The Persistence common object library component uses automake to build the library. Execute the following steps in order to build the component:

- autoreconf –vi
- ./configure
  - --with-database=”TheDatabase”
    - itzam
      - ➔ deprecated, not supported anymore
    - key-value-store
      - ➔ preferred database backend
  - optional with the following option
    - --enable-tests to enable the build of the tests for key-value-store backend (default)
    - --enable-debug to build with compile option "-g"
- make

- optional:
  - sudo make install install the library in the system

6. Testing and Logging

The test framework “check” has been used to write unit tests which will be run automatically when the test binary will be started. At the end a test report will be printed to the console showing first a summary about number of tests that have been executed and how many tests have been passed or failed. After the summary a test report will be generated showing the status of each test.

When a bug will be fixed a test will be written to verify the problem has been solved.

DTL (Diagnostic, Log and Trace) will be used by the PCO to send status and error. For details about DLT, please refer to the GENIVI DLT project page

(http://projects.genivi.org/diagnostic-log-trace/).

6.1 Running the Tests

There are unit tests available for the persistence key-value store component.

The unit tests are used to verify that the component is working correctly and exclude any regressions.

Please refer always to the source code to see the available tests.

Run the PCO test:

Change working directory to test/ folder

- run persistence key-value store unit test "/test_pco_key_value_store"

Expected results:

The expected result is to have 0 failures and 0 errors, see example output below:

```
100%: Checks: 22, Failures: 0, Errors: 0
test_pco_key_value_store.c:185:P:OpenlocalDB:test_OpenLocalDB:0: Passed
test_pco_key_value_store.c:211:P:OpenRCT:test_OpenRCT:0: Passed
test_pco_key_value_store.c:418:P:SetDataLocalDB:test_SetDataLocalDB:0: Passed
test_pco_key_value_store.c:504:P:GetDataLocalDB:test_GetDataLocalDB:0: Passed
test_pco_key_value_store.c:1162:P:SetDataRCT:test_SetDataRCT:0: Passed
test_pco_key_value_store.c:1312:P:GetDataRCT:test_GetDataRCT:0: Passed
test_pco_key_value_store.c:1376:P:GetDataSize:test_GetDataSize:0: Passed
test_pco_key_value_store.c:1638:P:DeleteDataLocalDB:test_DeleteDataLocalDB:0: Passed
test_pco_key_value_store.c:1783:P:DeleteDataRct:test_DeleteDataRct:0: Passed
test_pco_key_value_store.c:605:P:GetKeyListSizeLocalDb:test_GetKeyListSizeLocalDB:0: Passed
test_pco_key_value_store.c:754:P:GetKeyListLocalDb:test_GetKeyListLocalDB:0: Passed
test_pco_key_value_store.c:871:P:GetResourceListSizeRct:test_GetResourceListSizeRct:0: Passed
test_pco_key_value_store.c:1035:P:GetResourceListRct:test_GetResourceListRct:0: Passed
test_pco_key_value_store.c:2124:P:CacheSize:test_CacheSize:0: Passed
test_pco_key_value_store.c:1900:P:CachedConcurrentAccess:test_CachedConcurrentAccess:0: Passed
test_pco_key_value_store.c:2286:P:BadParameters:test_BadParameters:0: Passed
```

```
test_pco_key_value_store.c:2426:P:RebuildHashtables:test_RebuildHashtables:0: Passed
test_pco_key_value_store.c:2558:P:RecoverDatablocks:test_RecoverDatablocks:0: Passed
test_pco_key_value_store.c:2644:P:LinkedDatabase:test_LinkedDatabase:0: Passed
test_pco_key_value_store.c:322:P:ReadOnlyDatabase:test_ReadOnlyDatabase:0: Passed
test_pco_key_value_store.c:2731:P:MultipleWrites:test_MultipleWrites:0: Passed
test_pco_key_value_store.c:2843:P:WriteThrough:test_WriteThrough:0: Passed
```

The output above may vary as the test cases will be adopted or extended

7. Appendix

Appendix 1: Related Documents

Document	Description	Version	Link
Persistence Client Library User Manual	User Manual	2.5	http://docs.projects.genivi.org/persistence-client-library/1.0/GENIVI_Persistence_Client_Library_UserManual.pdf
Persistence Architecture Documentation	Architecture Documentation	1.8	http://docs.projects.genivi.org/persistence-client-library/1.0/GENIVI_Persistence_ArchitectureDocumentation.pdf

Appendix 1 - Related Documents