Mentor® Automotive

User Guide
Persistence Client Library

Version 2.6

<16/01/2016>

Content

List of Tables & Figures

1. Introduction

1.1 Purpose

The scope of this document covers interface description, building the library, component testing and debugging.

The document covers also some basic persistence concepts used to implement Persistence Client Library.

1.2 Revision History

Revision	Change	Date
V 1.0	Setup of Document	22.10.2013
V 1.1	Update after review	06.11.2013
V 1.2	Update after review (split content into two different documents, a user manual and a architecture manual)	25.11.2013
V 1.3	Added section default data handling	11.12. 2013
V 1.4	Added section roadmap of file API	14.01.2014
V 1.5	Added section best practices	17.01.2014
V 1.6	Added section to explain notification use; added configure switch --enable-pasinterface	29.01.2014
V 1.7	Added section **Fehler! Verweisquelle konnte nicht gefunden werden.** Backup and recovery" and **Fehler! Verweisquelle konnte nicht gefunden werden.** "Caching Strategy"	01.04.2014
V 1.8	Added section **Fehler! Verweisquelle konnte nicht gefunden werden.** ource Code Documentation	13.05.2014
V 1.9	Added section **Fehler! Verweisquelle konnte nicht gefunden werden.** ackup Location and updated section **Fehler! Verweisquelle konnte nicht gefunden werden.** lugin interface (changed configuration file format)	26.06.2014
V 2.0	Added section **Fehler! Verweisquelle konnte nicht gefunden werden.**	08.08.2014
V 2.1	Updated section **Fehler! Verweisquelle konnte nicht gefunden werden.** ow to build and **Fehler! Verweisquelle konnte nicht gefunden werden.** Testing and Debugging	11.09.2014
V 2.2	Added section **Fehler! Verweisquelle konnte nicht gefunden werden.** ystem Shutdown and added section **Fehler! Verweisquelle konnte nicht gefunden werden.** Configurable default data	09.12.2014
V 2.3	Added section 1.7 Application verification	21.02.2015
V 2.4	Updated section “2. System Shutdown” and “10.1.1 Plugin Types”. Added chapter “**Fehler! Verweisquelle konnte nicht gefunden werden.** esource Configuration Table”	27.03.2015
V 2.5	Complete rework and switched to new template	30.10.2015
V 2.6	Changed document license to CC BY-SA 4.0	16.01.2016

Table 1 - Revision History

1.3 Abbreviations & Terminology

Abbreviation	Description
ACID	Atomicity, Consistency, Isolation, Durability, see https://en.wikipedia.org/wiki/ACID
API	Application Programming Interface
DLT	Diagnostic Log and Trace
eMMC	embedded Multimedia Card
GENIVI	The GENIVI Alliance is a non-profit consortium whose goal is to establish a globally competitive, Linux-based operating system, middleware and platform for the automotive in-vehicle infotainment (IVI) industry.
IPC	Inter-Process Communication
ldbid	Logical Database Identifier
Linux VFS	Linux Virtual File System (Abstraction layer on top of a more concrete file system)
NAND	flash memory, a type of non-volatile computer memory
NOR	NOR Flash Memory, a type of non-volatile computer memory
NSM	Node State Manager (GENIVI lifecycle Component)
PAS	Persistence Administration Service
RCT	Resource Configuration Table
PCL	Persistence Client Library
SSD	Solid-state drive
PCL	Persistence Client Library
SSD	Solid-state drive
PCL	Persistence Client Library
PoC	Proof of Concept
PCO	Persistence Common Object http://git.projects.genivi.org/?p=persistence/persistence-common-object.git
Lbid	Logical Database Identifier

Table 2 – Abbreviations & Terminology

2. Persistence

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.

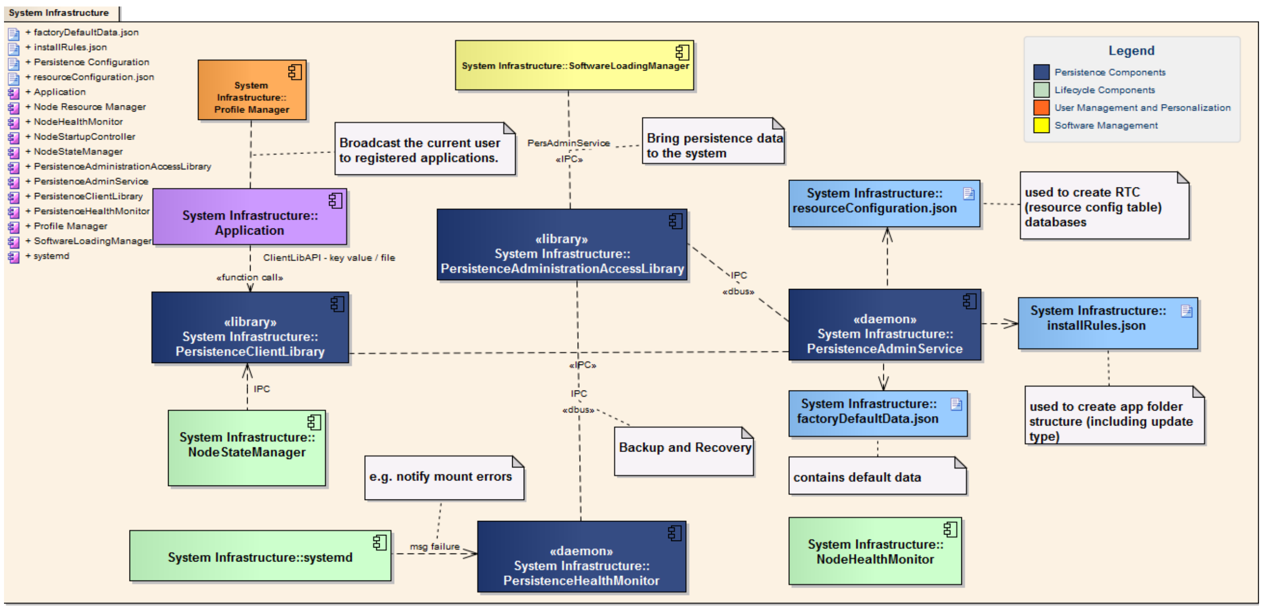

System Infrastructure
+ factoryDefaultData.json
+ installRules.json
+ Persistence Configuration
+ resourceConfiguration.json
+ Application
+ Node Resource Manager
+ NodeHealthMonitor
+ NodeStartupController
+ NodeStateManager
+ PersistenceAdministrationAccessLibrary
+ PersistenceAdminService
+ PersistenceClientLibrary
+ PersistenceHealthMonitor
+ Profile Manager
+ SoftwareLoadingManager
+ systemd
System Infrastructure:: Profile Manager
System Infrastructure::SoftwareLoadingManager
Broadcast the current user to registered applications.
PersAdminService
«IPC»
Bring persistence data to the system
Legend
Persistence Components
Lifecycle Components
User Management and Personalization
Software Management
System Infrastructure:: Application
ClientLibAPI - key value / file
«function call»
«library» System Infrastructure:: PersistenceAdministrationAccessLibrary
System Infrastructure:: resourceConfiguration.json
used to create RTC (resource config table) databases
IPC
«dbus»
«library» System Infrastructure:: PersistenceClientLibrary
«daemon» System Infrastructure:: PersistenceAdminService
System Infrastructure:: installRules.json
«IPC»
IPC
IPC
«dbus»
System Infrastructure:: NodeStateManager
System Infrastructure:: factoryDefaultData.json
used to create app folder structure (including update type)
Backup and Recovery
e.g. notify mount errors
contains default data
System Infrastructure::systemd
msg failure
«daemon» System Infrastructure:: PersistenceHealthMonitor
System Infrastructure:: NodeHealthMonitor

Figure 1 - System Context

2.1.2 Persistence Components Overview

The image below shows all persistence components and their relation.

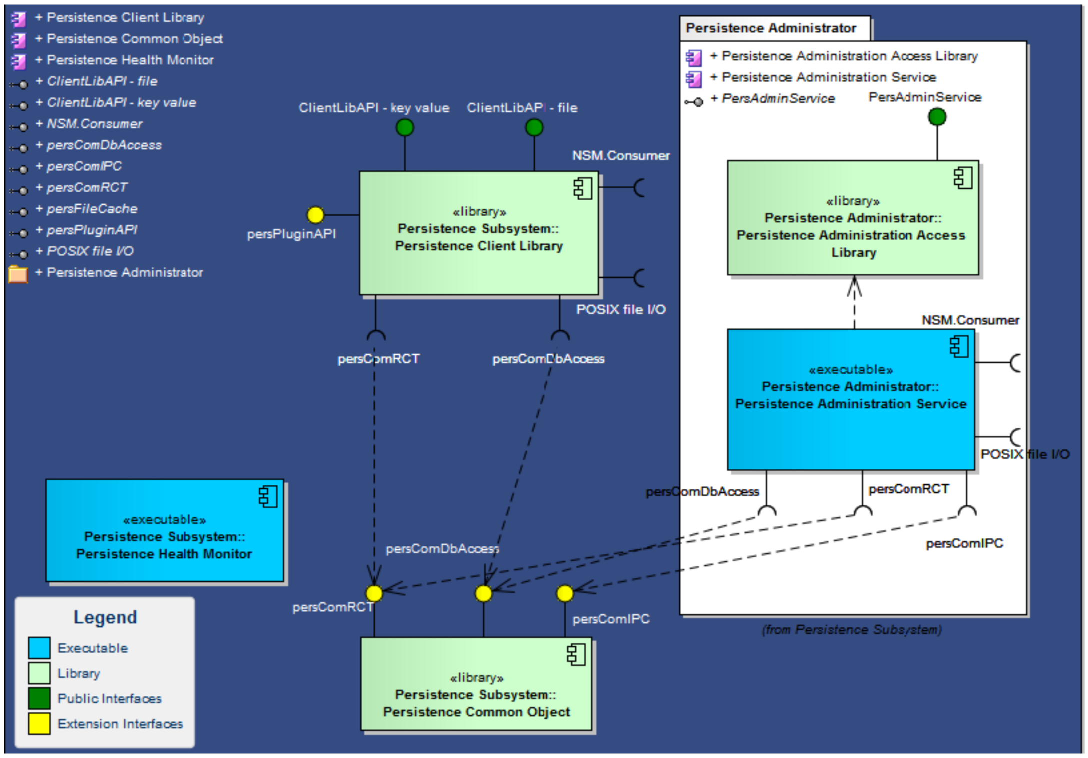

+ Persistence Client Library
+ Persistence Common Object
+ Persistence Health Monitor
+ ClientLibAPI - file
+ ClientLibAPI - key value
+ NSM.Consumer
+ persComDbAccess
+ persComIPC
+ persComRCT
+ persFileCache
+ persPluginAPI
+ POSIX file I/O
+ Persistence Administrator
ClientLibAPI - key value
ClientLibAPI - file
NSM.Consumer
persPluginAPI
«library» Persistence Subsystem:: Persistence Client Library
POSIX file I/O
persComRCT
persComDbAccess
Persistence Administrator
+ Persistence Administration Access Library
+ Persistence Administration Service
+ PersAdminService
PersAdminService
«library» Persistence Administrator:: Persistence Administration Access Library
NSM.Consumer
«executable» Persistence Administrator:: Persistence Administration Service
POSIX file I/O
persComDbAccess
persComRCT
persComIPC
(from Persistence Subsystem)
«executable» Persistence Subsystem:: Persistence Health Monitor
persComDbAccess
persComRCT
persComIPC
«library» Persistence Subsystem:: Persistence Common Object
Legend
Executable
Library
Public Interfaces
Extension Interfaces

Figure 2 - Persistence Components Overview

2.2 Persistence Client Library

GENIVI's Persistence Client Library, known as PCL, is responsible for handling persistent data, including all data read and modified during a lifetime of an infotainment system. "Persistent data" is data stored in a non-volatile storage such as a hard disk drive or FLASH memory

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 goal of standardization of data storage on the GENIVI platform is so that applications 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 the storage 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.

2.2.1 Responsibilities

The responsibility of the Persistence Client library is:

- Shared data management
- Persistent data may be accessed by different applications
- Local data management
- Persistent data is accessible only by the "owned" application
- Specific data management
- Concept can be extended for custom solution for early, secure, factory settings a.s.o storages / databases
- Local and shared file management
- Data can be stored in a file. The file can be accessed by different applications or only the "owned" application

2.2.2 Dependencies

In order to provide more flexibility the storage mechanism for persistence data has been moved into a separate component called Persistence Common Object.

Persistence Administration Service also uses this component to access persistent data.

2.2.3 Persistence Client Library License

The Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file.

You can obtain one at http://mozilla.org/MPL/2.0/.

2.2.4 Persistence Client Library Project Information

The Persistence Client Library is a GENIVI open source project and will be hosted by GENIVI.

Persistence Client Library project page:

- http://projects.genivi.org/persistence-client-library

Persistence Client Library repository:

- http://git.projects.genivi.org/?p=persistence/persistence-client-library.git

3. Persistence Concepts

3.1 Data Separation

Persistence data of each application will be located in a different database. Persistence data is also separated into different types of data.

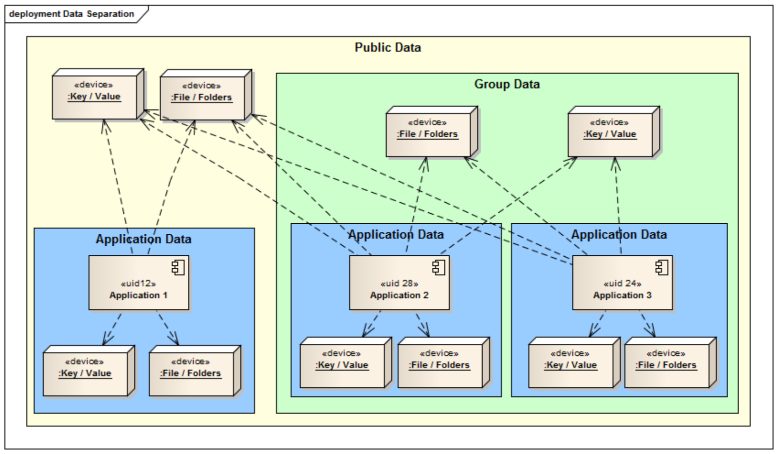


Figure 3 - Data Separation

3.1.1 Data Types and Logical Database ID

- **Local data**
  - Access is limited to the application itself
- **Local merged data**
  - Data from different applications that have been merged from different processes into only one single process
  - The key-value data is merged from different databases into one database
  - The incubated application accesses the local application data environment whereby the key are preceded by a prefix corresponding to the merged logical database identifier.
- **Group shared data**
  - Shared by a group of applications
  - Every member of the group has read access
  - There is only ONE master who has write access
- **Public shared data**
  - Shared by any application within the system
  - Every application within the system has read access
  - There is only ONE master who has write access

The so called logical database ID will be used to request access to the different data types.

- Local data
  - Use value 0xFF
- Shared data
  - Use value range 0x01..0x7F for different groups
- Public shared data
  - Value 0x00

For local and public data there a also defines in the file persistence_client_library.h available

- #define PCL_LDBID_LOCAL 0xFF /*!< value to access application local managed data */
- #define PCL_LDBID_PUBLIC 0x00 /*!< value to access public shared data, data is under control by the resource configuration in system */

3.1.2 Shared data access

Read and write access to group shared data and public shared data will be handled differently.

Every application with appropriate rights is able to read group or public data, but only one application will be able to write (modify) shared data.

All applications within a group will be able to read shared data of this group. All applications in the system will be able to read public data.

Only one application in a group will be able to write (modify) a resource and only one application in the system will be able to write (modify) a resource.

For each resource the application who is allowed to write (modify) this data will be assigned via the resource configuration table (RCT).

The application with the application ID written in the entry "*responsible*" in the RCT will be allowed to write (modify) the data.

This is valid for key-value and file data.

3.1.3 File System Representation

The data will be stored in different databases and files distributed over the file system.

The following image shows how the file/folder hierarchy should look like. From the file system entry point there are first in the hierarchy folders containing the different type of data, local, local merged, public and group data.

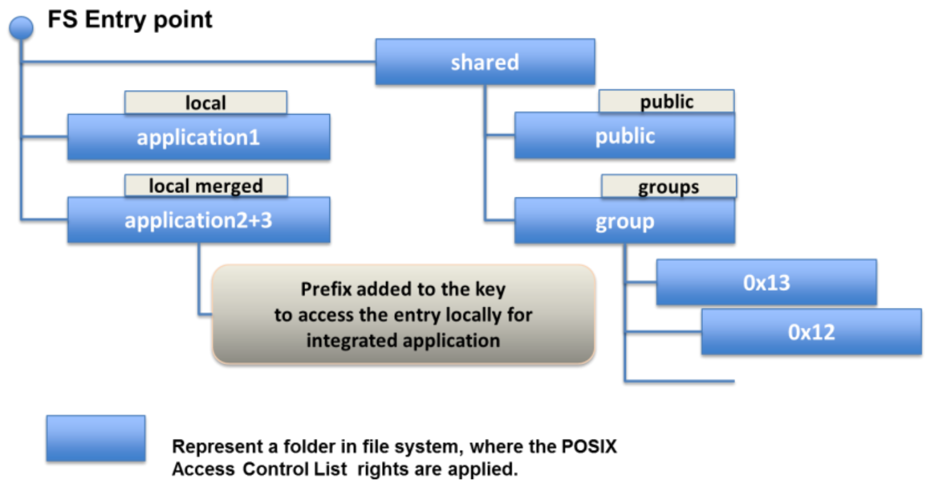
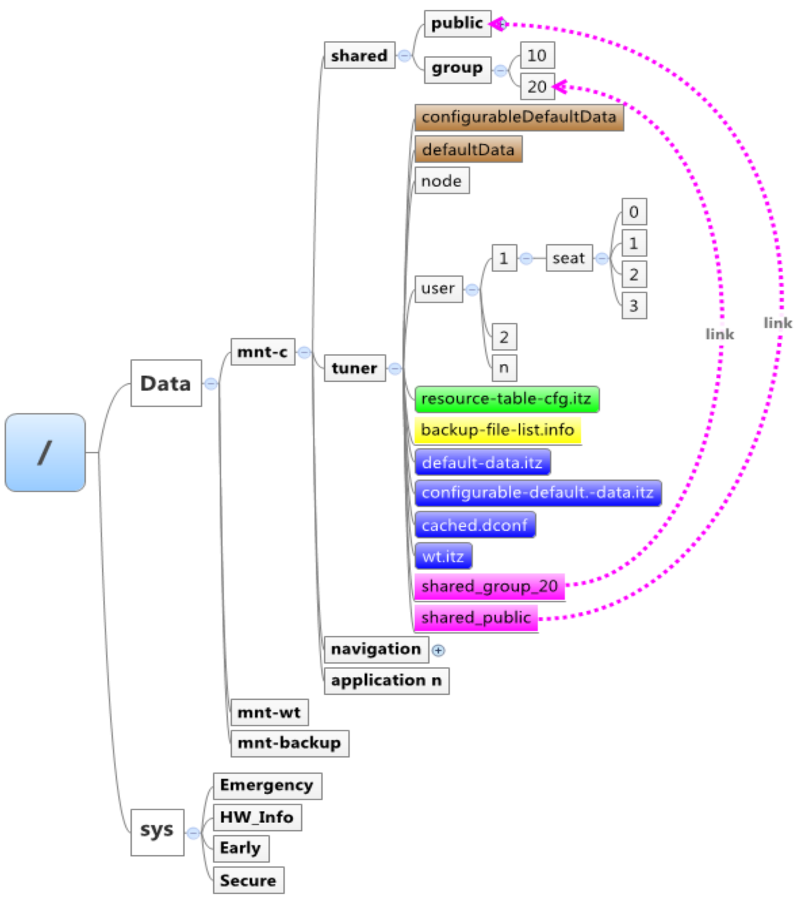

FS Entry point
shared
local
application1
local merged
application2+3
public
public
groups
group
0x13
0x12
Prefix added to the key to access the entry locally for integrated application
Represent a folder in file system, where the POSIX Access Control List rights are applied.

Figure 4 - File System Overview

In detail the file system hierarchy looks like the following image.



- **mnt-c**
  - cached file system partition
- **mnt-wt**
  - write-through partition
- **mnt-backup**
  - backup partition
- **Green**
  - RCT
    - Read-only/ read/write
    - Access rights
    - Storage (local/shared)
    - Resource type (key/file)
- **Blue**
  - Application user data
  - Default data
  - Configurable default data
- **Magenta**
  - Links to shared application data
- **Yellow**
  - Backup information
    - Create backup
    - Don‘t create backup

Figure 5 - File System Details

3.2 Interface Concept

The idea is to provide a simple and easy to use API to the application developers where they are able to quickly read and write persistent data.

The interface provides a schema less key-value store API as well as a file API where applications are able to store data in files using their own data format.

Versioning of persistent data will not be provided by the persistence subsystem, the application itself is responsible to store any version information in conjunction with the data. The reason not to provide this functionality is that only the application itself has this information. The versioning schema may differ between various applications and application supplier. Also when an application will be updated, the version schema may change including the data structure, this would also result in an update of the persistence client library to support the changed versioning.

The Logical database ID (ldbid) says what kind of data the access requested on, this means local, public or shared (public or group) data.

The Resource ID is the name of the requested resource; it is not needed to be a system-wide unique string.

The User No and Seat No will be provided by user management to identify the user and seat the data is requested for.

The table below gives an overview about the API parameters.

Logical DB ID (visibility / name space / location)			Resource ID	User No	Seat No
	single persistence client	**multiple persistence client**	**const char*** [path + key]	**unsigned int**	**unsigned int**
single application	**Local** (0xFFFF)	groupID			
multiple application	groupID	**Public** (0x0)			
unsigned int				node = 0x00 (no user)	no seat = 0x00

Table 3 - Interface Concept Details

3.2.1 User Number and Seat Number

The GENIVI user management has a component called profile manager which provides an API to applications to register for user and seat information. For details about this components please refer to the GENIVI user management concept in the GENIVI wiki. For details about the profile manager, see profile manager proof of concept component, see http://git.projects.genivi.org/?p=user-management/profile-manager.git.

If there is no component available in the system providing user and seat information, the persistence client library can still be used using one single static user number and seat number. It is recommended to use as user number the number one (user_no=0 cannot be used as user-ID because ‘0’ is defined as System/node) and for seat the seat number zero.

Note: In this case data for different users can't be identifies.

3.3 Security

Persistence does not provide an independent security concept. Security is in the system responsibility and persistence needs to be integrated into this concept.

Nevertheless persistence provides access control to persistence data in order prevent data to be accessed by applications not allowed to.

Due to the principle of data separation it is possible to apply different Linux user rights to the different subfolders of the applications. The application will be execute with the corresponding Linux user rights.

The same concept applies for the shared data, where the access restriction will be done using different Linux group ID's.

Note one or more applications can be started with the same POSIX UID.

The Persistence Administration Service is responsible to setup the default file and folder structure with the corresponding access rights (Linux user and group ID).

The following table shows an example how the group management can be configured.

Data	Owner	group
App1	App1	App1
App2	App2	App2
App3	App3	App3
Group		
App1gp1	App1	App1gp1
App1gp2	App1	App1gp1
App1gp3	App1	App1gp1
App2gp1	Ap2	App2gp1
App3gp1	App3	App3gp1

Table 4 - Linux Group Management

If an application needs access to the data shared by app1 in the sub group1 then this application needs to be added to the specific group app1gp1.

The logical database ID is the resolution of the name of the group over the system.

Note:

The setup of users and running the different applications with a special is in the responsibility of the using project.

PCL and/or PAS are not creating any user in the system.

To enhance access control also mandatory access control (e.g. SMACK) can be used.

3.4 Resource Configuration Table

The Resource Configuration Table (RCT) contains configuration details about every persistence resource in the system. It is one of the security aspects implemented in the client library as it allows detailed configuration of a resource.

The RCT will be created by the Persistence Administration Service (PAS) during system update process using JASON XML files. For more details about this please refer to the PAS documentation.

The PCL user normally doesn't care about the RCT, but should keep in mind that this configuration database exists and a resources behavior can be configured in the way described below.

System integrators are normally responsible for creating the resource configuration.

The configuration details are:

- **Storage policy**
  - Is the resource a cached or a write through resource
- **Storage type**
  - Is the resource a local, shared or custom
- **Resource type**
  - Is the resource key-value or a file
- **Permission**
  - Is this resource accessible read only or a read/write
- **Max storage size**
  - Responsible application
- **Name of the responsible application (master application).**
  - Only this application is allowed to change the value of the resource.
- **Custom name**
  - The custom name of this resource is a custom resource
- **Then custom id**
  - An ID. This could also be used as an index into and array if a custom array or has map will be used to store the resource

When an application tries to read/write a resource the RCT is always invoked as it is described in the following sequence:

- Application wants to write data
  - Query the resource table
  - RCT contains information about the requested resource
    - Check configuration
      - key-value or file
      - Size does not exceed configured max size
      - correct permission (write to a read-only resource)
    - If the configuration matches, write data to configured location
  - RCT does NOT contain information about the requested resource
    - Create entry about the resource in the RCT
    - Configure it as a local cached resource
    - ➔ write data

If there is no information about the requested resource, an application will nevertheless able to store this data with the restriction that the resource will be stored in the local cached path.

Note:

Only the persistence administration service or the responsible application is allowed to change the resource configuration table.

3.4.1 Cached and write-through Resources

Depending on the used storage media (e.g. flash drives) there are a limited number of write cycles possible. Typically around 100000 program/erase cycles wear-out of the flash device begins. Due to this circumstance writing of data needs to be controlled, because writing too many times data to a memory device may cause a SHORTER lifetime of the device.

Caching will be needed to reduce the number of data written to the device, so most of the resources are cached resources.

But there are also requirements to write back data directly to the memory device when the data has been changed (e.g. destination address for navigation, because if a reset occurs the user don't need to enter the address again).

To cover this requirement there is the possibility to setup a resource as a write-through resource.

Be careful with the write-through resources, as described above excessive usage may harm the flash memory devices.

3.5 API usage

Interface prototype:

int **pclKeyWriteData**(

unsigned int **ldbid**,

const char* **resource_id**,

unsigned int **user_no**,

unsigned char **seat_no**,

unsigned char* **buf**,

unsigned long **size**)

Persistence Client API	Persistence lib (from client resource table)	data base "input"	Comment
0xFF, "/pos/last pos", 0, 0	"/Data/mnt-c/Appl-1/cached.itz"	"/Node/pos/last pos"	local value (last position)
0, "/language/cur_lang", 3, 0	"/Data/mnt-wt/Shared/Public/wt.itz"	"/User/3/language/cur_lang"	public shared user value (current_language)
0xFF, "/status/op_doc", 3, 2	"/Data/mnt-c/Appl-1/cached.itz"	"/User/3/Seat/2/status/op_doc"	local user seat value (open_document)
20, "/address/home_addr, 4, 0	"/Data/mnt-c/Shared/Group/20/cached.itz"	"/User/4/address/home_addr"	group shared user value (home address)
0xFF, "/pos/last sat", 0, 0	"/Data/mnt-wt/Appl-1/wt.itz"	"/Node/pos/last satellites"	local value (last satellites)
0x84, "/links/last link", 2, 0	"/Data/mnt-wt/Appl-2/wt.itz"	"/84/User/2/links/last link"	local value (last link)
0xFF "/media/mediaDB.db", 1, 1	"/Data/mnt-c/Appl-1/"	"/User/1/Seat/1/mediaDb.db"	local file

Table 5 - API usage example

The resolution from the given API parameters to the key-value data or the file is done in done the following way:

- The lbid parameter gives a hint which resource configuration table (RCT) to use
  - First the RCT in the cached path will be queried
  - If the resource can be found in the RCT the storage policy will be queried
  - If storage policy is cached use the database in the cached path otherwise in the write through path.
  - 0xFFFF ➔ application local data, use the RCT in the application folder
  - 0x0 ➔ public data, use RCT in the shared folder
  - E.g. group 20 ➔ use the RCT in the group folder of group 20
- Then the input "key" to the database will be generated using resourceId, user_no and seat_no
  - User 0 and seat 0 ? it is NODE data ? input string is /Node/resourceID
  - User 3 and seat 2 ?it is USER data bound to a SEAT? /User/3/Seat2/resourceID

3.5.1 Example

```
int ret = 0;
unsigned int shutdownReg = (PCL_SHUTDOWN_TYPE_FAST |
PCL_SHUTDOWN_TYPE_NORMAL);
unsigned char buffer[READ_SIZE] = {0};


pclInitLibrary("Navigation", shutdownReg);


/**
 * Logical DB ID: 0xFF with user 1 and seat 1
 *       ==> local value accessible by all users (user 1, seat 1)
 */
ret = pclKeyReadData(0xFF, "pos/last_position", 1, 1, buffer, READ_SIZE);
memset(buffer, 0, READ_SIZE);


/**
 * Logical DB ID: 0xFF with user 0 and seat 0
 *       ==> local value accessible by all users (user 1, seat 1)
 */
ret = pclKeyReadData(0xFF, "language/country_code", 1, 1, buffer, READ_SIZE);


/**
 * Logical DB ID: 0x84 with user 2 and seat 1
 *       ==> shared user value accessible by A GROUP (user 2 and seat 1)
 *
 *       ==> used for shared testing
 */
ret = pclKeyWriteData(0x84, "links/last_link2",  2, 1, (unsigned char*)"Test
notify shared data", strlen("Test notify shared data"));


pclDeinitLibrary();
```

Then provided unit tests located under tests/ persistence_client_library_test.c is also a good code reference how to use the API.

3.6 Backup and Recovery Strategy

A backup and recovery strategy for data accessed by the file API is available in the Persistence Client Library.

There is no backup and recovery strategy for data accessed by the key-value API in the Persistence Client Library. The database implemented in the Persistence Common Object must provide this functionality.

A backup copy of the file including the generation of a checksum file is created when the file is modified the first time (Copy-on-write). There is no backup created if the file will only be read.

The checksum file is used during recovery to ensure that the backup copy is not corrupt.

A backup copy is created only once during the lifecycle and is removed during the shutdown phase when an application has successfully written the data using pclFileWriteData and calls writing the file back to non-volatile memory device (e.g. eMMC).

When writing the file back to e.g. eMMC the backup and checksum files are deleted. This is an indicator that the system has been shut down correctly. If the PCL detects in a new lifecycle that there is a backup and/or a checksum file available this is an indicator that the system hasn't been shut down correctly and the files must be checked for consistency.

Note:

A system failure which ends up in a restart of the system can occur at any time within a lifecycle.

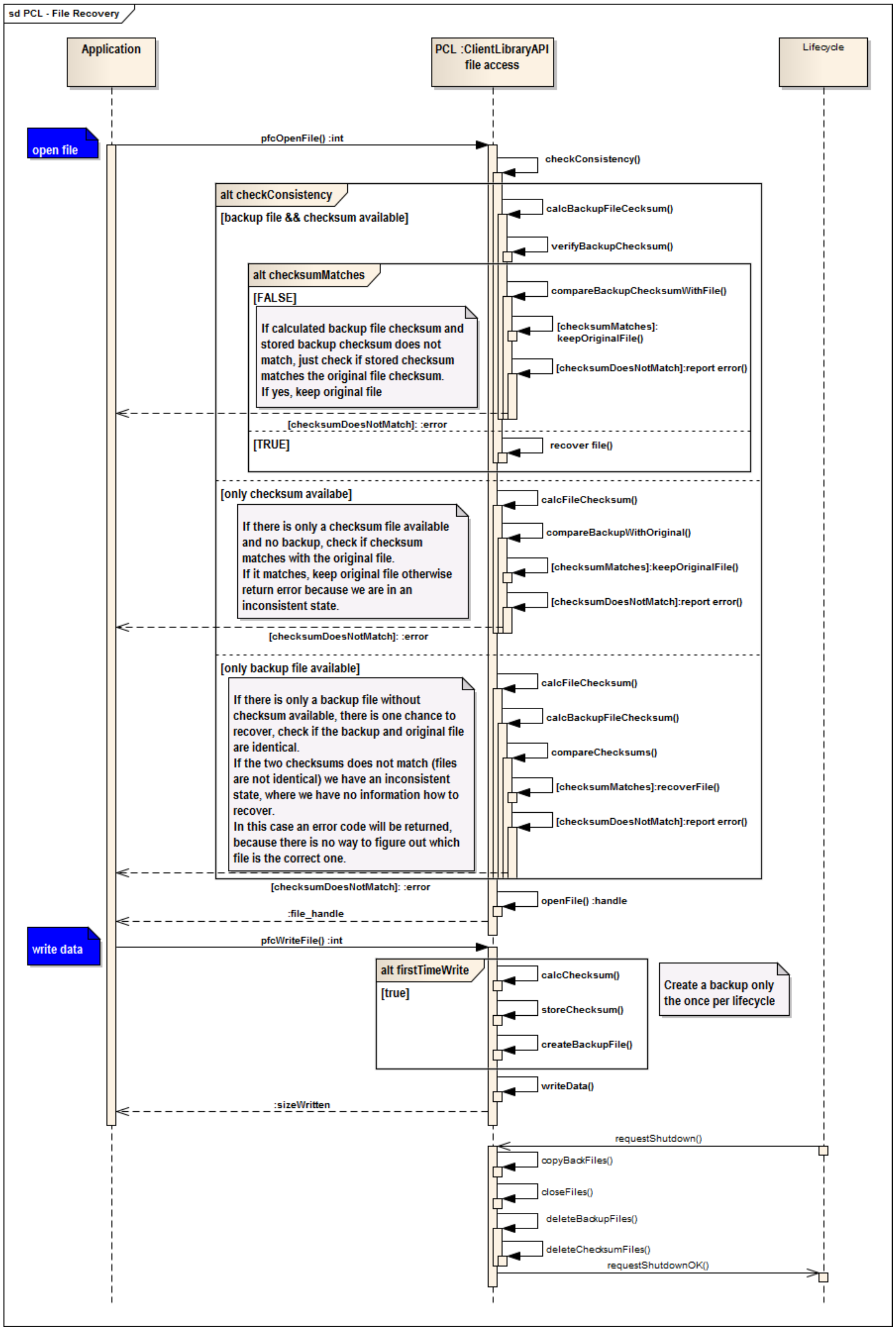
One critical path is during the pclFileWriteData function call, where the checksum is calculated and stored and the backup file is created.

The other critical path in the shutdown phase, where the backup file and the checksum files are deleted and the file is transferred from the RAM to the non-volatile storage device.

To cover this "reset" scenarios in the best way, there are some assumptions on the pclFileOpen function, where we try to detect such system failures and do a best guess to react properly. The following sequence diagram explains the recovery algorithm.

3.6.1 Recovery Algorithm

The sequence diagram below shows how the recovery mechanism works for files which are accessed via the file API.



sd PCL - File Recovery
Application
PCL :ClientLibraryAPI file access
Lifecycle
open file
pfcOpenFile() :int
checkConsistency()
alt checkConsistency
[backup file && checksum available]
calcBackupFileCecksum()
verifyBackupChecksum()
alt checksumMatches
[FALSE]
compareBackupChecksumWithFile()
If calculated backup file checksum and stored backup checksum does not match, just check if stored checksum matches the original file checksum. If yes, keep original file
[checksumMatches]: keepOriginalFile()
[checksumDoesNotMatch]:report error()
[checksumDoesNotMatch]: :error
[TRUE]
recover file()
[only checksum availabe]
calcFileChecksum()
If there is only a checksum file available and no backup, check if checksum matches with the original file. If it matches, keep original file otherwise return error because we are in an inconsistent state.
compareBackupWithOriginal()
[checksumMatches]:keepOriginalFile()
[checksumDoesNotMatch]:report error()
[checksumDoesNotMatch]: :error
[only backup file available]
calcFileChecksum()
If there is only a backup file without checksum available, there is one chance to recover, check if the backup and original file are identical. If the two checksums does not match (files are not identical) we have an inconsistent state, where we have no information how to recover. In this case an error code will be returned, because there is no way to figure out which file is the correct one.
calcBackupFileChecksum()
compareChecksums()
[checksumMatches]:recoverFile()
[checksumDoesNotMatch]:report error()
[checksumDoesNotMatch]: :error
openFile() :handle
:file_handle
write data
pfcWriteFile() :int
alt firstTimeWrite
[true]
calcChecksum()
storeChecksum()
createBackupFile()
Create a backup only the once per lifecycle
writeData()
:sizeWritten
requestShutdown()
copyBackFiles()
closeFiles()
deleteBackupFiles()
deleteChecksumFiles()
requestShutdownOK()

Figure 6 - Recovery Algorithm

3.6.2 Exclude form backup

It is possible to exclude files from the backup mechanism. For this reason there is the file BackupFileList.info in each application folder. Add the relative path and filename to this file and it will be excluded from the backup creation.

Example:

BackupFileList.info location:

/Data/mnt-c/lt-persistence_client_library_test/BackupFileList.info

File content:

/media/mediaDatabase.db

File to Exclude:

/Data/mnt-c/lt-persistence_client_library_test/user/X/seat/Y/media/mediaDatabase.db

Note:

The file will be excluded from the backup for every user on each seat.

3.7 Data Change Notification

The persistence client library provides a notification mechanism to inform applications if a key has been created, changed or deleted by another application. Registering to a notification is only possible on shared data.

A notification mechanism on files is not provided by the PCL, it is recommended to use inotify to monitor file system events the get the information.

3.7.1 How the Notification is working

An application interested in notifications about the status change of particular key of another application can register itself to get this information.

The key API provides two functions, pclKeyHandleRegisterNotifyOnChange and pclKeyRegisterNotifyOnChange.

Both functions have the parameter pclChangeNotifyCallback_t callback, which is a function pointer to a callback function.

This callback function will be called whenever the key under monitor changes its status, this means the key's content will be changed or it will be created or deleted.

The notification function's parameter pclNotification_s notifyStruct can be used to determine the kind of change.

The used IPC mechanism between the applications is dbus.

3.7.2 Example Code

Receiver Side:

```
// implement a notification callback function which will
// be called whenever a change notification
// will be received
int myChangeCallback(pclNotification_s * notifyStruct)
{
   printf("Notification received ==> lbid: %d | resource_id: %s |
         seat: %d | user: %d | status: %d \n", notifyStruct->ldbid,
         notifyStruct->resource_id,
         notifyStruct->seat_no,
         notifyStruct->user_no,
         notifyStruct->pclKeyNotify_Status );

   return 1;
}
…
// Registration works only on group or public data
ret = pclKeyRegisterNotifyOnChange(0x20, "links/last_link2", 2/*user_no*/,
1/*seat_no*/, &myChangeCallback);
```

Sender Side:

```
// Just write data and receiver will be notified
ret = pclKeyWriteData(0x20, "links/last_link2",  2, 1, dataBuffer,
strlen(dataBuffer));
```

4. System Shutdown

During system runtime all modified persistence data (key-value items and files) will be cached if it is not configured as a write-through. When the system is going to be shut down the modified key-value items and files will be written back from cache to the non-volatile memory device.

4.1 Shutdown Controlled by PCL

In the preferred shutdown scenario the write back of the data is under control of persistence client library (PCL).

An application using the PCL calls the function pclInitLibrary, providing the information which shutdown notifications the PCL shall receive (use PCL_SHUTDOWN_TYPE_FAST or PCL_SHUTDOWN_TYPE_NORMAL for the shutdownMode parameter).

Calling this function registers the PCL to the GENIVI Node State Manager (NSM) as a shutdown client in order to receive shutdown notifications.

Then the application itself must also register itself to the NSM to receive shutdown notifications.

When a shutdown occurs the NSM will send first a shutdown notification to the Application.

The application writes data for the last time of the current lifecycle to the PCL.

Then the PCL receives also the shutdown notification form the NSM and writes the modified data back to the non-volatile memory device.

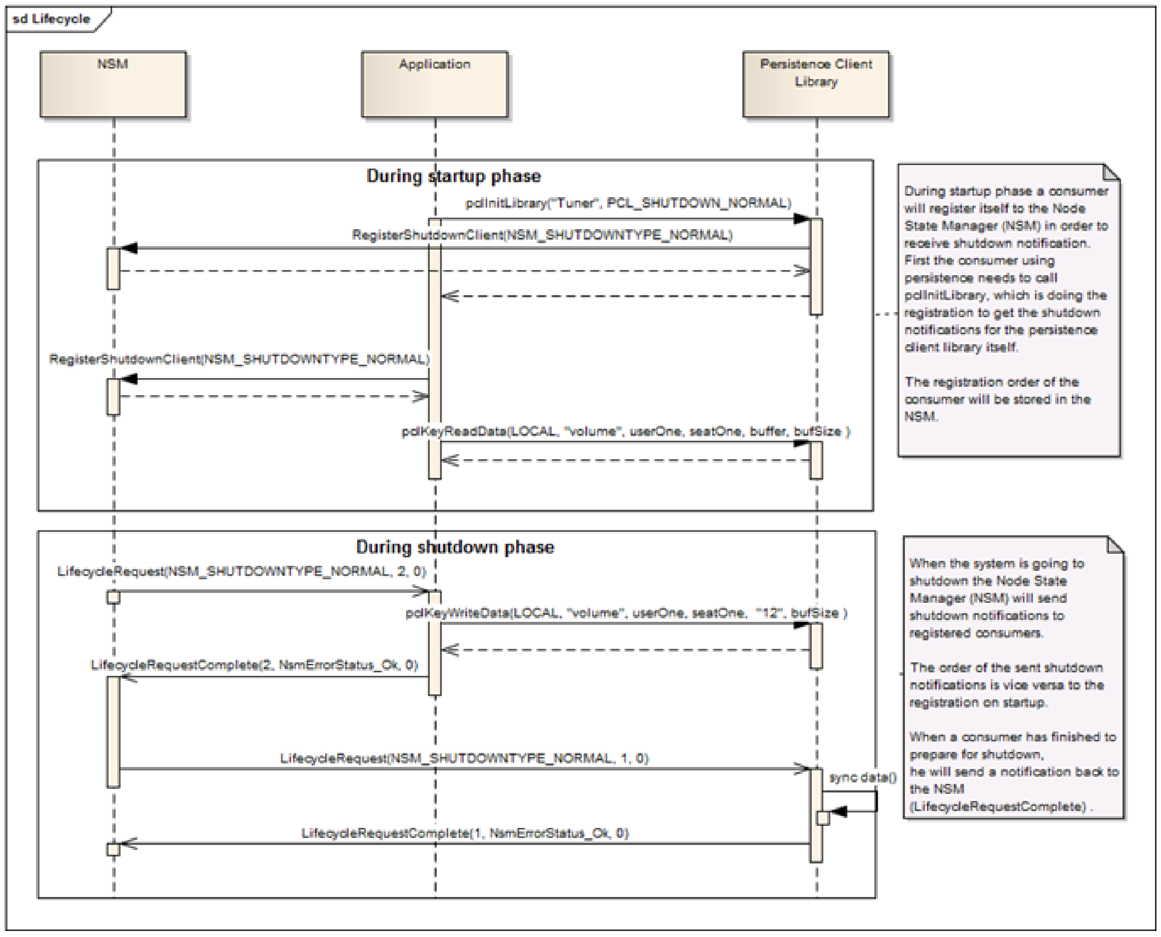

sd Lifecycle
NSM
Application
Persistence Client Library
During startup phase
pclInitLibrary("Tuner", PCL_SHUTDOWN_NORMAL)
RegisterShutdownClient(NSM_SHUTDOWNTYPE_NORMAL)
RegisterShutdownClient(NSM_SHUTDOWNTYPE_NORMAL)
pclKeyReadData(LOCAL, "volume", userOne, seatOne, buffer, bufSize)
During startup phase a consumer will register itself to the Node State Manager (NSM) in order to receive shutdown notification. First the consumer using persistence needs to call pclInitLibrary, which is doing the registration to get the shutdown notifications for the persistence client library itself.
The registration order of the consumer will be stored in the NSM.
During shutdown phase
LifecycleRequest(NSM_SHUTDOWNTYPE_NORMAL, 2, 0)
pclKeyWriteData(LOCAL, "volume", userOne, seatOne, "12", bufSize)
LifecycleRequestComplete(2, NsmErrorStatus_Ok, 0)
LifecycleRequest(NSM_SHUTDOWNTYPE_NORMAL, 1, 0)
sync data()
LifecycleRequestComplete(1, NsmErrorStatus_Ok, 0)
When the system is going to shutdown the Node State Manager (NSM) will send shutdown notifications to registered consumers.
The order of the sent shutdown notifications is vice versa to the registration on startup.
When a consumer has finished to prepare for shutdown, he will send a notification back to the NSM (LifecycleRequestComplete) .

Figure 7 - Lifecycle Interaction

4.1.1 Example Code

The shutdown controlled by persistence client library is the most convenient way for applications as the just red to call pclInitLibrary and at the end of the lifecycle pclDeinitLibrary.

```
// register PCL for “normal shutdown” events.
unsigned int shutdownReg = PCL_SHUTDOWN_TYPE_NORMAL;

// provide a system unique application name to the PCL
pclInitLibrary(“mediaplayer”, shutdownReg);

// register app to the NSM to receive shutdown events
…
// wait for shutdown events distributed by the NSM
…
// shutdown event has been received, finally write data to PCL

// some key-value data
ret = pclKeyWriteData(PCL_LDBID_LOCAL, "currentTitleNumber", 3, 2,
keyValueDataBuf, strlen(keyValueDataBuf));

// some file data
size = pclFileWriteData(fd, fileDataBuf, strlen(fileDataBuf));

// now trigger PCL to write back modified key-value and file data
rval = pclLifecycleSet(PCL_SHUTDOWN);

// now call deinit function
pclDeinitLibrary
```

4.2 Shutdown Controlled by the Application

Another scenario has been integrated to the PCL the GENIVI lifecycle management is not available for some reasons. The application using the PCL calls the function **pclInitLibrary** with the **PCL_SHUTDOWN_TYPE_NONE** for the **shutdownMode** parameter.

Now the PCL did not register itself as a shutdown client to the NSM and did not get any shutdown notification.

The application has the full control when to write back modified persistent data to the non-volatile memory device. To write back the data the application uses the function **pclLifecycleSet** passing PCL_SHUTDOWN as shutdown parameter. The PCL now write back the modified data, and blocks any further write to persistence.

If a cancel shutdown (or restart) will be done, persistence can be unblocked using pclLifecycleSet passing **PCL_SHUTDOWN_CANEL.** Reading/Writing data from/to persistence is now possible again.

Attention:

In order to prevent the misuse of this shutdown scenario the call of **pclLifecycleSet** with **PCL_SHUTDOWN_CANEL** is limited to 3 times per lifecycle.

4.2.1 Example Code

```
// register PCL not to get any shutdown notifications
unsigned int shutdownReg = PCL_SHUTDOWN_TYPE_NONE;

// provide a system unique application name to the PCL
pclInitLibrary(“mediaplayer”, shutdownReg);

// register app to the NSM to receive shutdown events
…

// wait for shutdown events distributed by the NSM
…
// shutdown event has been received, finally write data to PCL

// some key-value data
ret = pclKeyWriteData(PCL_LDBID_LOCAL, "currentTitleNumber", 3, 2,
keyValueDataBuf, strlen(keyValueDataBuf));

// some file data
size = pclFileWriteData(fd, fileDataBuf, strlen(fileDataBuf));
```

5. Default Data Handling

Persistence has two different types of default data, the factory default data and the configurable default data.

5.1 Reading Default Data

The PCL has no special interface to read and write default data.

If an application wants to read the data of a resource the PCL is doing this in the following way:

- first try read the data for the resource, if available return this data
- if no data is available, check if configurable default data available and return this data
- if no configurable default data is available, check if there is factory default data is available, and return this data.
- If still no data is available return an error.

5.2 Writing Default Data

Factory default data can't be changed, but configurable default data can be adjusted.

Setting data back to factory default data or to configurable data can't be done using the PCL API, instead the Persistence Administration Service API needs to be used.

For more details see PAS API documentation, especially the following function:

```
long persAdminDataRestore(PersASSelectionType_e type,
                          PersASDefaultSource_e defaultSource,
                          const char* applicationID,
                          unsigned int user_no,
                          unsigned int seat_no);
```

5.3 Configurable default Data

Configurable default data can be modified using the PCL file and/or key-value API.

To modify configurable default key-value data use the function pclKeyHandleWriteData and pclKeyWriteData.

If configurable default data for files needed to be modified use the pclFileWriteData function (first open the file using the pclFileOpen function).

In order to modify configurable default data use the define PCL_USER_DEFAULTDATA for the user_no parameter. Configurable default data is user and seat independent.

6. Persistence Client Library API

6.1 API Details

The Persistence Client Library provides a C-API for applications allowing loading and storing persistent data. The API is divided into two main parts, the key-value API and the file API.

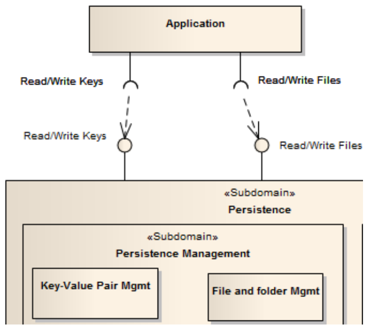


Figure 8 - PCL API

The Persistence Client Library offers two sorts of functions for the Key-value and file interface to read and write data, handle functions and non-handle functions. For the handle functions first the open function (pclKeyHandleOpen) needs to be called, and then using the handle the corresponding read/write (e.g. pclKeyHandleReadData) function can be called.

If requested to just read or write only once to a resource non-handle functions are also available (e.g. pclKeyReadData).

A file will not be identified directly by a file system path when using for example the pclFileOpen function. The logical database ID, the resource ID, the user ID and the seat ID will be translated into a path and a filename within the file system hierarchy. The same is valid for a resource stored via the key-value API. The final resource name in the database is a concatenation of the resourceID, the user ID and the seatID.

6.2 Datatype

The PCL does not support storing of any complex data structures or any other data types than unsigned char*.

It was a design decision not to support data type families like e.g. glib or Qt data types or other complex custom data structures as they all can be stored using the unsigned char* data type. Serialization/deserialization or casting must be done by the using application.

6.3 Persistence Data Versioning

Persistence Client Library does not support any versioning or migration of data. It is in the responsibility of the using application to keep track of the data versioning and migration. An application should store version information together with the data by itself.

6.4 Doxygen documentation

For details of the file API, please refer to the doxygen documentation.

The doxygen package must be installed on the system.

Run "doxygen doc/pcl_doxyfile" in the persistence-client-library folder and the html documentation will be generated in the doc folder.

7. Setup Target Environment

7.1 File System Partition

Persistence Client Library needs a file system partition to store the data. The partition needed to be mounted to **/Data/mnt-c** and **/Data/mnt-wt**.

It is requested to have one persistence partition which will be mounted to the two folders stated above (one partition will be mounted to two different mointpoint).

It is also recommended to create a separate partition for backup data which must be mounted to **/Data/mnt-backup**

Example:

sudo mount -t fsType /dev/sdx /Data/mnt-wt

sudo mount -t fsType /dev/sdx /Data/mnt-c

7.2 Components needed by Persistence Subsystem

Some other GENIVI components are required to be installed on the system.

- GENIVI Lifecycle components are available on the system
    - http://projects.genivi.org/node-state-manager/
    - http://projects.genivi.org/node-startup-controller/
    - Make sure Node State Manager is running
        - Start "node-state-manager"
- Persistence Administration Service is available in the system and running
    - The GENIVI Persistence Administration Service is available here:
        - http://git.projects.genivi.org/?p=persistence/persistence-administrator.git
- Persistence Common Object is available in the system
    - The GENIVI Persistence Common Object is available here:
        - http://git.projects.genivi.org/?p=persistence/persistence-common-object.git
    - use the key-value-store backend
        - run configure step with "-- with_database=key-value-store"
- Persistence partitions has been created and mounted and test data has been setup
    - If data partition and data is not available, see 7.1
    - Use the Persistence Administrator Service to setup the test data.
        - make sure node-state-manager and pers_admin_svc are running
        - Now install the test data:
            - "persadmin_tool install /path_to_test_data/PAS_data.tar.gz"
        - The test data is provided by the PCL (../test/data/PAS_data.tar.gz)
- Make sure dlt-daemon is running
- Make sure D-Bus system bus is available

7.3 Persistence Plugins

Make sure persistence plugin configuration file is setup correctly and installed into the intended folder. For more details see next chapter 8 Plugin Interface

8. Plugin Interface

In order to keep the used storage back-end flexible Persistence Client Library provides a plugin interface. With this interface it is possible to easily integrate storage different.

The plugin (shared object) will be loaded according the information in the configuration file, either it will be loaded when the function **pclInitLibrary** will be called or when a plugin function will be called the first time. The loading behavior is configured in the configuration file.

The configuration file providing the information about the available plugins is expected to be in the following default location: "/etc/pclCustomLibConfigFile.cfg".

The default location and filename can be changed using the environment variable "PERS_CLIENT_LIB_CUSTOM_LOAD".

For each key-value resource a different custom storage solution can be applied using the Persistence Resource Configuration Table (RCT). The attribute storage type needs to be set to "custom" and the name of the plugin shared object needs to be added to the attribute custom name.

The Persistence Common Object will be used as default storage plugin.

8.1 Plugin API

The plugin API for custom storage back-ends is describe in the header "**persistence_client_custom.h**". The plugin API for the default storage (Persistence Common Object) is described in the header "**persComDbAccess.h**" of Persistence Common Object.

8.2 Plugin Types

There are predefined custom plugin types available:

- **default**
  - default plugin used for general storing if no custom type is set in the RCT
  - Persistence Common Object must be used for this plugin (libpers_common.so)
- **early**
  - plugin for loading early data
- **secure**
  - plugin to implement encrypting
- **emergency**
  - plugin for emergency
- **hwinfo**
  - plugin to query hardware information
- **custom1**
  - used to implement something different then the predefines
- **custom2**
  - used to implement something different then the predefines
- **custom3**
  - used to implement something different then the predefines
- **default**
  - used for default storage mechanism implementation.

As default plugin the Persistence Common Object (libperscommon) API must be implemented by this plugin.

8.3 Loading Type

When will the plugin be loaded?

- init
    - plugin will be loaded when the pclInitLibrary function will be called
- od
    - on demand loading, the plugin will be loaded when a function of the plugin will be called the first time.

8.4 Initialization Type

Specify the init function which will be called.

- sync
    - use the synchronous init type
- async
    - use the asynchronous init type

8.5 Configuration File Format

The format of the plugin configuration file is the following:

<predefined type><blank><path to plugin><blank><loading type><blank><init type>

Example:

```
hwinfo /usr/local/lib/libhwinfoperscustom.so init async
secure /lib/libsecureperscustom.so od async
custom3 /usr/local/lib/libcustom3perscustom.so od sync
emergency /usr/local/lib/libemergencyperscustom.so init sync
```

8.5.1 Adjusting Configuration File.

The configuration file will be generated using autotools with the correct library path.

Nevertheless this configuration file located in config/pclCustomLibConfigFile.cfg severs as an example only and needs to be adjusted according to the projects environment and needs including the availability of the plugins in the system.

9. How To Build

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

9.1 Dependencies

The client library has the following dependencies

- Components
    - automotive-dlt
        - http://projects.genivi.org/diagnostic-log-trace/
    - Persistence Common Object (libpers_common.so)
        - http://git.projects.genivi.org/?p=persistence/persistence-common-object.git
        - Add "-- with_database=key-value-store" to the configure step
    - dbus
    - check
        - unit test framework for C
        - needed when configured with "--enable-tests" to run unit tests
- Tools
    - autotools
    - libtool

9.1.1 Persistence Common Object

The Persistence Common Object (libpers_common.so) is the default plugin to read/write persistent data managed by the key-value API.

Persistence Common Object uses Itzam/C as default database, which is currently not supported anymore. It is recommended to use the key-value-store backend instead.

Use "-- with_database=key-value-store" for the configure step.

9.2 Building the Library

The Persistence Client Lib component uses automake to build the library.

Execute the following steps in order to build the component:

- autoreconf –vi
- configure, with the following options
    - –enable-tests
        - to enable the build of the tests
    - --enable-pasinterface
        - enable the PAS interface (disabled by default)
    - --enable-appcheck
        - to enable application check
- make
- optional:
    - sudo make install

10. 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 Persistence Client Library to send status and error.

For details about DLT, please refer to the GENIVI DLT project page (http://projects.genivi.org/diagnostic-log-trace/).

10.1 Running the Tests

There are unit tests available for the persistency client library component available.

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 tests:

- run persistency unit test "./persistence_client_library_test"

Expected results:

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

```
./persistence_client_library_test

Running suite(s): Persistency client library

100%: Checks: 13, Failures: 0, Errors: 0

persistence_client_library_test.c:141:P:GetData:test_GetData:0: Passed

persistence_client_library_test.c:357:P:SetData:test_SetData:0: Passed

persistence_client_library_test.c:400:P:SetDataNoPRCT:test_SetDataNoPRCT:0:
Passed

persistence_client_library_test.c:434:P:GetDataSize:test_GetDataSize:0: Passed

persistence_client_library_test.c:478:P:DeleteData:test_DeleteData:0: Passed

persistence_client_library_test.c:236:P:GetDataHandle:test_GetDataHandle:0:
Passed

persistence_client_library_test.c:652:P:DataHandle:test_DataHandle:0: Passed

persistence_client_library_test.c:727:P:DataHandleOpen:test_DataHandleOpen:0:
Passed

persistence_client_library_test.c:578:P:DataFile:test_DataFile:0: Passed

persistence_client_library_test.c:604:P:DataFileRecovery:test_DataFileRecovery
:0: Passed

persistence_client_library_test.c:794:P:Cursor:test_Cursor:0: Passed

persistence_client_library_test.c:864:P:ReadDefault:test_ReadDefault:0: Passed

persistence_client_library_test.c:886:P:ReadConfDefault:test_ReadConfDefault:0
: Passed
```

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

10.2 Quickstart

Normally the default file and folder structure including the default data as well as the RCT and the database will be setup by the Persistence Administration Service.

If a user wants to extend the provided test application or create an own test application it can be done without using the PAS to extend the file and folder structure.

An example could be a navigation application likes to use the PCL it can be done in the following way:

Example Code:

```
int ret = 0;
unsigned char buffer[READ_SIZE]  = {0};
unsigned int shutdownReg = PCL_SHUTDOWN_TYPE_FAST | PCL_SHUTDOWN_TYPE_NORMAL;

// initialize PCL
pclInitLibrary("navigation", shutdownReg);

ret = pclKeyWriteData(0xFF, "mapSettings", 1, 2, "northUp",
strlen("northUp"));
ret = pclKeyReadData(0xFF, "mapSettings",  1, 2, buffer, READ_SIZE);

// do something

// end the usage of PCL
pclDeinitLibrary();
```

Steps to proceed:

- Use the PCL initialization function and provide the name of the application the PCL should use. The example above uses the identifier "navigation"
- Create a folder with the identifier name as used for the PCL init. Here the folder must be named "navigation".
  - In this example the path must look like /Data/mnt-c/navigation/
- Copy the existing RCT and database from the folder lt-persistence_client_library_test provided in the test data
  - Copy RCT file "resource-table-cfg.itz" into /Data/mnt-c/navigation
  - Copy the database "cached.itz" to /Data/mnt-c/navigation
- Now you should be able to write and read data like shown in the example above.
  - **Note:** As there is no entry for this resource in the RCT, the resource will be stored in the local cached path.
- If another application wants to use the PCL, e.g. a tuner application, just proceed step 1 to 3 using an appropriate name (e.g. tuner) as application name and folder

11. Guidelines and Best Practices

11.1 Shared Data Notification

In the persistence subsystem there is a mechanism to register on notification for shared data change. As there is the possibility to misuse this feature to bypass existing IPC in the system it is recommended not to use this mechanism for such bypass.

Use the existing IPC in the system for synchronization and data exchange.

11.2 Key-value Interface

The intention of the provided key-value interface is to set/get small data only, not Megabytes or even Gigabytes.

The default size limit for key-value data will be set to 16kByte by default.

11.3 Write-Through Data

Be careful about the resource that has the storage policy write-through. Use this only for resources that absolutely need to be written back to the memory device when they were changed. You need take into account the amount of data and the write frequency the write-through resource will be written.

Using this policy for to many resources it can reduce the memory device lifetime dramatically, as you may exceed the max number of program/erase cycles.

11.4 User and Group Rights

As written in the chapter 3.1 Data Separation the persistent data can be setup using different user and group rights per applications. Every application is running under a different user, and so it is guaranteed that the data can't be accessed by other applications.

If this policy can't be applied within your system, persistence will work but will lose this security aspect.

11.5 RCT customID

The customID member of the resource configuration table (RCT) could be used to speed up access time to resources within a custom storage extension plugin.

If a hash table or array will be used to store data in the custom extension plugin, the customID can be used to generate the hash value into the hash table or the index into the array offline and store this in the RCT for this resource.

If a customID is available the PCL generates a string using the lbid and the customID which then will be used as path parameter for the custom plugin interface functions.

This may speed up access time as there is no need to generate the hash value or the index during runtime, but the hash table and the array must be static and can't be changed during runtime.

11.6 Flash Lifetime Consideration

The proposed persistence solution is not a 100% bulletproof out of the box solution to prevent flash memory device wear-out.

Depending on the system setup like the specific used memory device and file system.

Every memory device (NAND, NOR or eMMC) is different according to the max number of program/erase cycles and the used wear-leveling algorithm (static or dynamic).

The system setup is specific to the project the PCL will be used in and completely independent. When using the persistence subsystem an investigation is needed about behavior of the specific system.

It's needed take into account the amount of data written, the write frequency, the max number of program/erase cycles of the chip and the used war-leveling algorithm.

Item	Static	Dynamic
Endurance	Longer life expectancy	Shorter life expectancy
Performance	Slower	Faster
Design Complexity	More complex	Less complex
Typical Use	SSDs	USB Flash Drives

Table 6 - Dynamic vs. static wear-leveling

11.7 SQLite

As there is a demand for more complex queries, e.g. for a media databases, support of other databases by the GENIVI persistence subsystem must be considered.

SQLite is a popular choice as database engine in embedded system and due to this circumstance we need to deal with this database within GENIVI.

As there are some concerns when using SQLite according flash lifetime a safe flash memory device usage must be guaranteed.

If SQLite will be used on flash memory devices we can't predict how often data will be written to the device. SQLite for example updates depending on the mode of operation the database for every access, also in read only operation mode, because of read transactions. Due to their architecture SQLite is creating also a very high read load on the file system also reduces the lifetime of the flash memory device.

Scenarios which are reducing flash lifetime:

- Writing to the same erase sector too often.
  - This problem is solved by wear leveling provided by the file system. File systems with both static and dynamic wear leveling do a better job than ones with just dynamic wear leveling. Flash devices typically have endurance between a few thousand cycles and 100,000 cycles per sector. Wear leveling spreads the cycles uniformly across the sectors and thus keeps the cycling within the endurance limit for a longer period, but can't guarantee that this will work for the lifetime of the car which is minimum 10 years.
- Writing to the same sector too quickly.
  - Here we are looking at the time between two erases of the same sector. This is related to the previous issue, but not the same. Here again, wear leveling will help.
- Reading from the file too often.
  - Reading from the same location too often can cause read disturbance and loss of bits. This can be solved by having a read cache in the file system.

Scenarios which are flash friendly:

Flash memory is organized in so called erase blocks, which typically have the size of 8 or 16 MB, depending on the memory size.

If data will be written to flash memory, an erase block must be deleted first.

Taking this into account data to flash memory should be written sequentially in sizes matching the erase block size, otherwise there is some write overhead.

This chapter summarizes different approaches for a save integration of SQlite into the persistence subsystem environment. Each approach should be carefully considered before it will be used if it fits into project requirements.

11.7.1 In-memory Database Approach

SQLite database is normally stores data in a single file on the non-volatile memory device. But it is also possible to store the database in RAM. This approach is able to prevent the issues with flash memory device as the database will be run in RAM.

Proposed sequence:

- Open a database in-memory
- Create a plain database table matching the table that is available in the file database
- Attach the file-based database to the in memory database (e.g. "ATTACH 'persistence.db' AS pers")
- Import data from file-based database into in-memory database (e.g. "INSERT into DataTable SELECT * from pers.DataTable")
- Add a trigger to the database schema for delete/insert/update
- Work on the in-memory database (e.g. "UPDATE DataTable SET VALUE = ...")
- Write back the deleted/insert/updated data according the set triggers when the system is going down
- Close database

Pro:

- Only deleted/insert/updated items will be written back

Cons:

- An in-memory database is a private database, only one connection is possible (not possible to share data)
- Is the attached database is bigger than the memory or bigger than the reserved memory spot, some additional management needs to be done (attaching only parts of the file-database ...).
- Not a transparent solution for legacy or OSS applications as the proposed sequence above must be used.
- ACID compliant only for the lifecycle in case of power loss

11.7.2 Ramdisk/tmpfs Approach

For this approach the SQLite database file will be copied into ramdisk or tmpfs. As the database will be run in the RAM there are no flash memory issues.

Proposed sequence:

- Copy database into a ramdisk or tmpfs
- Work on the database
- Copy the database back when the system is going to shutdown

Pro:

- Complete transparent for the application

Cons:

- Need enough memory to copy the complete database into ramdisk/tmpfs (also keep in mind there is the page cache where database pages will be kept)
- If there is not enough memory available, then the concept will not work.
- If only one entry has been deleted/inserted/updated during the lifecycle, the complete database needed to be copied back
- Could case shutdown delay.
- Transaction log needs also to be in ramdisk/tmpfs, ACID compliant only for the lifecycle in case of power loss

11.7.3 SQLite OS Backend (VFS)

"The OS Interface - also called the "VFS" - is what makes SQLite portable across operating systems.

Whenever any of the other modules in SQLite needs to communicate with the operating system, they invoke methods in the VFS.

The VFS then invokes the operating-specific code needed to satisfy the request. Hence, porting SQLite to a new operating system is simply a matter of writing a new OS interface layer or "VFS"."

This could be used to implement a VFS that is doing the data caching, see: http://sqlite.org/vfs.html

Pro:

- Access to the underlying file system is completely under the control of the VFS backend implementation

Cons:

- Probably the a complex solution to implement

11.7.4 Miscellaneous

As data should be written sequentially to increase memory lifetime, enabling “write-Ahead Logging” in SQLite should be considered.

Write-Ahead Logging.

The default method by which SQLite implements atomic commit and rollback is a rollback journal. Beginning with version 3.7.0, a new "Write-Ahead Log" option (hereafter referred to as "WAL") is available.

For more details please refer to the following page: https://www.sqlite.org/wal.html

12. Persistence Usability Rules

GENIVI proposed 3 level rules about the usability of the Persistence Client Library.

12.1 Complete Usage of the PCL

Only file API functions provided by the PCL will be used to access persistent data, except pclFileCreatePath and pclFileReleasePath functions.

Backup creation and recovery is within the responsibility of the PCL.

12.2 Partly Usage

Only the two API functions will be used to access persistent data.

The function pclFileCreatPath will be used to get a path to a file where the persistent data will be stored.

The application can access the file in every way the application wants. Either the application can use POSIX file access functions if the data will be written directly to a file or use SQL functions to open and access a database file.

Backup creation and recovery is within the responsibility of the PCL.

12.3 Free Access to Persistence Data

An application has free access to the specific structure defined in the persistence concept. It is allowed to directly access the persistence file/folder structure in the way the application wants. The complete management of backup and recovery is in responsibility of the application.

13. Appendix

Appendix 1: Related Documents

Document	Description	Version	Link
Persistence Client Library Component Specification	Component Specification	2.3.1	https://collab.genivi.org/wiki/download/attachments/1247269/GENIVI_Component_Specification_PersClientLibrary-2.3.1.docx
Persistence Architecture Documentation	Architecture Documentation	1.8	http://docs.projects.genivi.org/persistence-client-library/1.0/GENIVI_Persistence_ArchitectureDocumentation.pdf
Persistence Common Object Library User Manual	User Manual	2.2	http://docs.projects.genivi.org/persistence-client-library/1.0/GENIVI_Persistence_Common_Object_UserManual.pdf

Appendix 1 - Related Documents