Content

- Production Program - KPIT Role
- Software Platform Architecture
- GENIVI Components Adaptation
- GENEVI Adaptation – Persistency
- Persistency Adaptation Challenges
- Persistency Adaptation Approach
- Custom Tools / Utilities
- Problems Experienced
- Next steps
**Production Program - KPIT Role**

**KPIT as a Lead Software Integrator for MY18 & MY19 Production Program**

- Complete SW integration & release
- HMI and application development
- KIVI middleware porting and adaptation
- System Infrastructure development / Adaptation
- Linux BSP porting, tuning and optimization
- 3rd party software integration
- Cybersecurity, Fast Boot, FOTA
- Ownership of system KPIs
- Build management
- Integration Testing

**North American OEM**

**Japanese Tier-1**

**KPIT**

**3rd Party Supplier**
- Bluetooth, Wi-Fi
- Media
- Delta Update
- CarPlay
- Android Auto
- CarLife-Baidu
- Delta Update
- ECNC
- Quick-boot

**Specifications**

**Contract**

**Develop**

**Integrate**

```
Tier-1 Developed
Tuner
SXM
Vehicle / CAN Firmware
```

**3rd Party Supplier Specifications**

- H/W
- S/W

**KPIT Specifications**

- H/W
- S/W

**Integrate**

**Develop**
GENIVI Component Adaptation in KIVI

**Life Cycle Management**

- GENIVI provided LCM components are used almost as it is with minor bug fixes
  - Node State Manager (NSM)
  - Node Startup Controller (NSC)

**Persistency**

- GENIVI provided Persistency components are used almost as it is with minor bug fixes
  - Persistence Client Library (PCL v1.0.0) (user 1, seat 0), private data
  - Persistence Administrator (PAS v 1.0.5)
  - Persistence Common Object (PCO v1.0.3)

**Audio Manager**

- GENIVI Audio Manager framework used. All plugins are developed by KPIT as per production program requirement
  - Audio Manager Daemon (version 1.0)
  - Audio Manager Controller Plugin (version 1.0)
  - Audio Manager Routing Plugin (version 1.0)
  - Audio Manager Command Plugin (version 1.0)
Assumption: Understanding of Genivi components

We know

- Persistence Client Library
- Persistence Administrator
- What are Cacheable items, Write-through items.
- Persistence setup Json tar files
- Persistence folder structure
GENEVI Adaptation – Persistency
GENIVI Adaptation – Deployment
Persistency Adaption Challenges for Production

1. Database deployment on target
   - First time Deployment during EOL Testing
   - Deployment of newly added applications using software update
   - Deployment on host during development and testing

2. Serialization & De-serialization of key-value pair data

3. Use of same database for multiple applications as a part of single process

4. Enforcing cacheable and write-through behaviors.

5. Backup trigger management

6. Exception handling (File system corruption, mount failure,)

7. Custom tools requirement for development and testing
### 1.1 Approach – App deployment workflow

**Host Platform**

1. **Create Parameter XML (Eclipse)**
2. **Generator Tool**
   - **Xml to cpp & Json**
   - **Output**: `<<appName>>.Tar.gz`
   - **resource-configuration.json**
   - **default-data.json**
   - **default files**
3. **Json to Database Convertor**

**Target Platform**

1. **Client Application Code**
2. **Auto Generated .cpp/.h Files**
3. **Persistence Integration Application Client**
4. **Deployment**
   - **SetupPersistency**
   - **PAS**
   - **Database Placeholders**
5. **Output - Database Folder structure**
   - `<<appName_Data.tar.gz>>`
   - (Host Deployment)

**Target Platform Components**

- **K-Store**
- **PCL, PCO**
1.2 Approach - Database Deployment / Installation On Target

First time installation and through S/W update

• Rootfs contains application JSON tar files
• During first boot before application is fully operational respective database folder structure is created
• New application database is installed using software update.
2.1 Approach – Abstract Serialization & de-serialization

App view

Data types

string listsoflinks;

Infra view

KIVI type

kvariant_t(listoflinks);

PCL view

Buffer

unsigned Char* RawData;

DB files
2.2 Approach – Application uses primary data types

Application Persistables as Simple variables

```c
if( DEFAULT_APP_ID != start->appId )
{
    m_cachedData->HOME_factoryAppOrder[index].HOME_appIndex = start->appIndex;
    m_cachedData->HOME_factoryAppOrder[index].HOME_appId = start->appId;
    m_cachedData->HOME_factoryAppOrder[index].HOME_badgeState = DEFAULT_ZERO;
    m_cachedData->HOME_factoryAppOrder[index].HOME_iconState = HOME_IMG_APP_NORMAL;
    index++;
}

/*generate*/
kint32_t HOME_DBCheck = 0;
USER_STRUCT_CACHE1_DECLARE HOME_factoryAppOrder[26];
kuint16_t HOME_currentPage;
bool HOME_shouldReadDefault;
kint32_t HOME_calAppOrder[26];
bool HOME_firstRunStatus;
```

Developer handles basic data types
2.3 Activity - Persistency integration

- Create **Parameters.xml**
- Autogenerate **persistableparameters**
- Use persistibles in **normal code**
2.5 Snapshot – Cachables and Writethrough

```xml
<Struct Name="factoryAppOrder" Description="Default Application Loading Order"
    numberOfElements="26" Storage="local" Policy="cached" Permission="Read-Write">
    <UnsignedShort Name="appIndex"/>
    <Integer Name="appId"/>
    <UnsignedInteger Name="event"/>
    <UnsignedShort Name="badgeState"/>
    <UnsignedShort Name="iconState"/>
    <Boolean Name="isNative"/>
    <Boolean Name="isPersistence"/>
</Struct>

<Boolean Name="isDataSharingOn" Description="Status of data sharing flag" Storage="local" Policy="writethrough"
    Permission="Read-Write" value="true" UserSpecific="yes"> </Boolean>
```
2.4 Snapshot - Persistables in normal code.

**Instantiate the Persistence autogenerated classes.**

```cpp
// instantiate persistables
DECLARE_PERSISTENCE(m_Cached, ipc)
REGISTER_ERROR_HANDLER(m_Cached, myErrorHandler, NULL)
END_DECLARE_PERSISTENCE
```

**Load the Persistable data at startup**

```cpp
// Load the Persistables.
kstore_status_t status = m_cachedData->Load();
m_userAppOrder[location] = m_CachedData->HOME_factoryAppOrder[index];
// let us know which page the user was
m_currentHomePage = m_CachedData->HOME_currentPage;
```
2.4 Snapshot - Persistables in normal code.

Store the Persistable data at shutdown.

```c
kstore_status_t status = m_Cached->Store();
LOG5((TEXT("Store status-> %d\n"), status)); //LCOV_EXCL_LINE
if(status != KSTORE_STATUS_OK)
    LOGERR((TEXT("Store failed with status-> %d \n"), status));
status = m_Cached->HandleShutdown();
```

Destroy the Persistables

```c
HomeLogicManager::~HomeLogicManager() {
    RELEASE_PERSISTENCE
}
```
Sample code HMI

Sample_Src_HomeLogicManager_cpp.html

Sample_Media_file_usage_cpp.html

SampleFavoriteManager_FilePathType.cpp.html
2.5 Snapshot - Writethrough Persistables

Assign value and call macro to store

```cpp
m_pImmediateWriteData->isDataSharingOn = status;
PERSIST_WT_PARAM(m_pImmediateWriteData);
```

Macro in autogenerated code that calls explicit set.

```cpp
#define PERSIST_WT_PARAM(X)\
X->Store(); //macro with specific parameter to set.
```
2.6 Approach – Advantages

- Abstraction of persistence APIs.
- Abstraction of storage policy of persistency infrastructure.
- Automatic coupling to lifecycle.
- Json Tars with the app developer.
- Ease of trying out and testing.
- Less number of PCL Key-value pairs so low memory utilization
- Multiple applications within single process using same database
- Multiple applications under multiple team working for same process
- Conditional backup during shutdown
3.1 Approach – multiple xmls generate multiple json

App_projections.xml

```xml
<Boolean Name="CarplayAppState"
Description="Persists Value of AppSetting for Carplay"
Storage="local" Policy="cached" Permission="Read-Write"
value="true" UserSpecific="yes"/>
```

Resource-configuration.json

```json
{  "Projection_CarplayAppState":  {   "policy": "cached", "permission": "Read-Write",   "storage": "local", "max_size": "2", "responsible": "Me",   "custom_name": "none", "type": "key", "customID": "d38046 "  },
}
```

App_hmi_core.xml

```xml
<File Name="favoritesDB" Description="Favorites Database storage path"
Storage="local" Policy="cached" Permission="Read-Write"
UserSpecific="yes" MaxFileSize="7000" dataType="Path">
<Path>/usr/kpit/Persistence/favorites.db</Path></file>
```

Resource-configuration.json

```json
{  "COREFavoritesDB":  {   "policy": "cached", "permission": "Read-Write",   "storage": "local", "max_size": "7000", "responsible": "Me",   "custom_name": "none", "type": "file", "customID": "d38046 "  },
}
```
"config_appl" : "HMIF", "version" : "v1.0.0", "resources" :
{
  "Projection_DBCheck":
  {
    "policy": "cached", "permission": "Read-Only", "storage": "local",
    "max_size": "5", "responsible": "Me", "custom_name": "none",
    "type": "key", "customID": "d38046"
  },
  "Projection_CarplayAppState":
  {
    "policy": "cached", "permission": "Read-Write", "storage": "local",
    "max_size": "2", "responsible": "Me", "custom_name": "none",
    "type": "key", "customID": "d38046"
  },
  "CORE_DBCheck":
  {
    "policy": "cached", "permission": "Read-Only", "storage": "local",
    "max_size": "5", "responsible": "Me", "custom_name": "none",
    "type": "key", "customID": "d38046"
  },
  "CORE_favoritesDB":
  {
    "policy": "cached", "permission": "Read-Write", "storage": "local",
    "max_size": "7000", "responsible": "Me", "custom_name": "none",
    "type": "file", "customID": "d38046"
  }
}
3.2 Approach – Post build utility merges jsons

```json
{
  "config_appl": "HMIF", "version": "v1.0.0", "resources": {
    "Projection_CarplayAppState": {
      "policy": "cached", "permission": "Read-Write", "storage": "local", "max_size": "2", "name": "Projection_CarplayAppState", "custom_name": "none", "type": "key", "customID": "d38046"
    },
    "CORE_favoritesDB": {
      "policy": "cached", "permission": "Read-Write", "storage": "local", "max_size": "7000", "name": "CORE_favoritesDB", "custom_name": "none", "type": "file", "customID": "d38046"
    },
    "CORE_DBCheck": {
      "policy": "cached", "permission": "Read-Only", "storage": "local", "max_size": "5", "name": "CORE_DBCheck", "custom_name": "none", "type": "key", "customID": "d38046"
    }
  }
}
```
3.3 Approach – Multiple app keys under one process
4 Approach :- Backup trigger management

- Change in write through persistable is backed up soon after.
- Change of cacheable data registers application for backup.
- At shutdown setup persistence service backups up all the applications registered for backup.
<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Tools</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Packpersistencesetup</td>
<td>• Merging separate json tars of multiple applications into one under a process.</td>
</tr>
</tbody>
</table>
| 2      | Persistencexml2cpp           | • Autogenerating Persistable classes  
• Creating Json tars                                           |
| 3      | JsonToDatabase               | • To generate tar files that can be used for host testing.               |
Problem Experienced

Major problems experienced

- Data base corruption
- File and File System corruption
- Mount failure
- Schema update failed as a part of software update

System strategy

- Avoid abrupt shutdowns if possible
- Avoid file system unmount failures during shutdown
- Optimize data write events
- Recover from backup
- File system checks (fsck)
- Recreate default persistency
- Health Monitoring and failure detection mechanism
- Factory Reset
- System recovery
Next steps

• Enhancements
  – Serialization of Structure inside structures
  – Serialization of Structures strings as members.
  – Serialization of classes.
• Provision to store file buffer inside key-value pair to store file inside database
• Auto generation tool in Python instead of Java xtend
• Using persistence framework work on Android platform
Thank you!

Visit GENIVI at http://www.genivi.org or http://projects.genivi.org
Contact us: help@genivi.org

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 (CC BY-SA 4.0) 
GENIVI is a registered trademark of the GENIVI Alliance in the USA and other countries. 
Copyright © GENIVI Alliance 2018.