Stepping Stones for Java Card Applet Developers

Version 1.0

April 2024

Intellectual Property Rights (IPR) Disclaimer

Contents

1. References

Java Card 3.0.5	Java Card 3 Platform Virtual Machine Specification, Classic Edition Version 3.0.5	Nov 2017
	Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.5	Nov 2017
	Java Card API, Classic Edition Version 3.0.5	
Java Card 3.1.0	Java Card Platform Virtual Machine Specification, Classic Edition Version 3.1	Feb 2021
	Java Card Platform Runtime Environment Specification, Classic Edition Version 3.1	Feb 2021
	Java Card API, Classic Edition Version 3.1.0	
GSMA	SGP.22 RSP Technical Specification Version 3.0	October 2022
	SGP.05 Embedded UICC Protection Profile Version 4.1	March 2023
	SGP.25 Embedded UICC for Consumer Devices Protection Profile Version 1.0	05 June 2018
ETSI	ETSI TS 102 221 V16.6.0 (2021-10) Smart Cards; UICC-Terminal interface; Physical and logical characteristics (Release 16)	October 2021

	ETSI TS 102 226 v16.1.0 (2022-10) Smart Cards; Remote APDU structure for UICC based applications (Release 16)	October 2022
	ETSI TS 102 241 v16.2.0 (2021-08) Smart Cards; UICC Application Programming Interface (UICC API) for Java Card™ (Release 16)	August 2021
	ETSI TS 102 267 v16.0.0 (2021-08) Smart Cards; Connection Oriented Service API for the Java Card™ platform (Release 16)	August 2021
3GPP	3GPP 31 102 Technical Specification Group Core Network and Terminals; Characteristics of the Universal Subscriber Identity Module (USIM) application (Release 16)	December 2022
Global Platform	GlobalPlatform Card API (org.globalplatform) v1.7	July 2019
	GlobalPlatform Card Composition Model Security Guidelines for Basic Applications Version 2.0	November 2014
TCA	Integrated SIM: A Practical Approach	April 2022
	Recommended 5G SIM Technical Definition: Enhanced for 3GPP Release 17	December 2022
	Interoperability Stepping Stones Release 7	December 2009
	Interoperable Format Technical Specification v3.3.1	August 2023

2. Abbreviations

3GPP	3rd Generation Partnership Project (3GPP)
5G SA	5G Standalone
APDU	Application Protocol Data Unit
API	Application Programming Interface
APN	Access Point Name
CAP	Converted Applet
CAT-TP	Card Application Toolkit – Transport Protocol
CoD	Transient Clear on Deselect
eSIM	Embedded SIM (Subscriber Identity Module)
EEP	Electrically erasable programmable
ETSI	European Telecommunications Standards Institute
eUICC	Embedded UICC (Universal Integrated Circuit Card)
FCP	File control parameters
GSMA	Global System for Mobile Communications Association
ICC	Integrated Circuit Card
I/O	Input/output
IOT	Internet of things
iUICC	Integrated (UICC) Universal Integrated Circuit Card
JCVM	Java Card Virtual Machine
LTE	Long Term Evolution
M2M	Machine to machine
MNO	Mobile Network Operator
MVNO	Mobile Virtual Network Operator
NAA	Network Access Application
NVM	Non Volatile Memory
OEM	Original Equipment Manufacturer
OS	Operating System
OTA	Over the air
PIN	Personal identification number
QR	Quick-response
RAM	Random access memory
RE	Runtime Environment
RMI	Remote Method Invocation
RSP	Remote SIM Provisioning
SCP	Secure Channel Protocol
SDOs	Standardization organizations
SIM	Subscriber Identity Module
SM-DP+	Subscription management – data preparation (as in SGP.22)
SMS	Short Message Service
TLV	Tag-length-value
UE	User equipment
UI	User Interface
UICC	Universal Integrated Circuit Card
UST	USIM Service Table

3. Definitions

eSIM	eSIM is the generic term applied to devices and eUICCs that support Remote SIM Provisioning as defined by GSMA.
eUICC	A UICC which enables the remote and/or local management of profiles in a secure way that meet GSMA requirements for Remote SIM Provisioning and are certified in accordance to the GSMA compliance programme. The term originates from “embedded UICC”.
GlobalPlatform API	The GlobalPlatform API provides services to applications (e.g. cardholder verification, personalisation, or security services).
Java Card	Technology that allows Java-based applications to be run securely on smart cards and more generally on similar secure small memory.
Remote SIM Provisioning (RSP)	Specification developed by GSMA that allows consumers to remotely manage the Subscriber Identity Module (SIM) embedded in a device.
Subscriber Identity Module (SIM)	A generic term for the application(s) residing on the UICC that identifies a subscriber and allows them to securely access a mobile network (e.g. 4G or 5G). SIM is sometimes used interchangeably with the term UICC or SIM card.
Universal Integrated Circuit Card	The platform, specified by ETSI, which can be used to run multiple security applications. These applications include the SIM for 2G networks, USIM for 3G, 4G and 5G networks, CSIM for CDMA, and ISIM (not to be confused with integrated SIM) for IP multimedia services. UICC is neither an abbreviation nor an acronym.

4. Introduction: The Java Card Applet Stepping Stones

For several decades, the UICC has been leveraged to deliver advanced and innovative value-added services – which have been further enriched as the capabilities of UICC technology has evolved.

The emergence of eUICC technology is having a transformative impact, but also presents new interoperability challenges for stakeholders as, on a specific eUICC, several profiles containing applets by third parties are downloaded with minimal or no integration activity.

To ensure seamless integration and simplify deployments across the highly complex telecom ecosystem, Trusted Connectivity Alliance (TCA) – as a leading global industry association – has a proven record identifying and promoting the need for strong interoperability. Among myriad other initiatives, the publication of its various 'Stepping Stones' documents have played an important role in guiding industry stakeholders.

The publication of this latest Stepping Stones document marks a continuation of TCA's decades-long efforts to promote interoperability. It should also be noted that previous documents such as 'Trusted Connectivity Alliance: Interoperability Stepping Stones Release 7' - developed in December 2009 - are still valid and can be used as reference.

Overall, this document serves as an indispensable resource for Java Card applet developers, offering a holistic view of the technology's nuances, best practices, and security considerations to facilitate the creation of robust and interoperable solutions in the dynamic landscape of smart card development.

This document delves into various aspects of Java Card technology, providing a thorough exploration of the Interoperability Stepping Stones and the evolving applet ecosystem. The overview of Java Card versions, including Java Card 3.0.5 and Java Card 3.1.0, is accompanied by a summary of essential updates from ETSI and 3GPP, offering developers valuable insights into the ever-changing landscape. The guidelines for Java Card applet development, encompassing NVM, fields, local variables, and more, serve as a roadmap for best practices.

Furthermore, the document extends its focus to the critical realm of security, offering recommendations applicable to all applets and additional measures for sensitive ones. The inclusion of the TCA Loader, TCA eSIM Interoperability Service, and insights into interoperability events, such as the Interoperability Test Fest, underscores the importance of seamless integration.

For developers seeking practical guidance, the document concludes with a comprehensive checklist, ensuring that applet developers are well-equipped to navigate the intricacies of interoperability.

4.1. Target Audience

The target audience are Java Card applet developers interested in developing interoperable applets targeting Secure Elements and SIM/USIM applications from Trusted Connectivity Alliance members and, in particular, for devices that are used in Remote SIM Provisioning (RSP) environments. The document covers common pitfalls and ways to avoid them, improving the overall quality of applets.

4.2. Problem Statement

The focus of the current version of the document is to provide guidelines for Java Card applet developers to help prevent issues that could result in incorrect operation of and/or permanent damage to the SIM/eSIM/eUICC/iUICC/integrated eUICC. The guidelines become even more important in the context of the eUICC and iUICC considering that, in case of permanent damage, the eUICCs cannot be replaced like the traditional removable UICCs. (For more information on the eUICC/iUICC technology and ecosystem, you can refer to TCA document 'Integrated SIM: A Practical Approach').

4.3. Notes for Readers

UICCs (independent of their form-factor) with capability of managing the subscriptions via Remote SIM Provisioning (RSP), referred to as eUICCs in the GSMA specification, are commonly referred to as eSIMs in the commercial world.

The GSMA specifies the Subscription Management solutions for three different markets - M2M, IoT and Consumer, with each having a dedicated technical specification. While the guidelines that apply to the applets that are developed for all three market variants broadly remain the same, the applets would differ in the functionality and services catering to the particular eUICC variant.

5. Understanding the Java Card Applet Ecosystem

This section is intended to provide Java Card applet developers with an overview of key Java Card versions, including Java Card 3.0.5 and Java Card 3.1.0. This is accompanied by a summary of essential recent updates from GSMA and 3GPP to help developers understand and navigate the evolving ecosystem.

5.1. Java Card Technology (JC 3.0.5, JC 3.1.0,…)

Java Card technology, tailored for smart cards and memory-constrained devices, has undergone significant evolution.

Beginning with Java Card 2.2.1 (2003) and progressing through versions like Java Card 3.0 (Classic Edition, 2008) and Java Card 3.1 (2018), the technology has embraced improvements such as Java Card RMI, extended APDUs, new APIs for Certificate Management and support for new cryptographic algorithms.

The latest version, Java Card 3.2, introduced in 2023, expands the horizon with support for advanced security algorithms, improved cryptographic operation performance, and compatibility with diverse card form factors.

Developers should understand the capabilities of the various Java Card versions and, to enhance interoperability with in-field eUICCs, choose the minimal Java Card version required by the developed service. For example, if an applet is compiled using the Java Card 3.1.0 APIs, it will not be compatible with a Java Card 3.0.5 eUICC. However, a Java Card 3.0.5 applet can be downloaded on both 3.0.5 and 3.1.0 eUICCs.

This applies in general to all the packages that are imported by the application. For example, An application importing Release 6 of the ETSI TS 102 241 API can be loaded on Release 15 cards, while an application importing Release 15 of the ETSI TS 102 241 API cannot be loaded on Release 6 cards. This means that to enhance interoperability, application developers should always choose the lowest version of the API required by the application services.

It should be noted as a reference, GSMA SGP.22 consumer eUICCs v2.x requires at least Java Card 3.0.4, while the ETSI Release18 APIs require at least Java Card 3.1.0.

5.2. GSMA and 3GPP: Understanding Major Updates for Java Card Developers

GSMA has driven the standardisation of the eUICC in all variants; in particular, in the SGP.22 v2.4 several evolutions relevant to applet developers have been introduced.

Some of the key features of SGP.22 v2.4:

- Remote provisioning: eSIMs can now be provisioned remotely, without the need for physical access to the device.
- Remote management: eSIMs can now be managed remotely, including updating the firmware and profiles.
- Support for new applications: SGP.22 v2.4 supports a wider range of applications for eSIMs, including 5G, IoT, and M2M.

In 3GPP, the Rel-17 has furtherly enhanced The 5G-SA architecture, with a significant evolution versus 4G/LTE. It is based on a set of independent functions being deployed in a cloud infrastructure. The full 5G architecture promises to allow MNOs and other mobile service providers to build systems that can offer innovative services that can generate enormous benefits.

5G-SA, with its complex micro-services topology in a cloud environment, is a combination of information technology and communication technologies. It has brought change to the network architecture, which enables the network to flexibly support a variety of application scenarios. Those changes raise different security requirements and distinct security configurations for the network, especially for network deployment and operations.

Due to compatibility issues and need to offer a continuity of service to users, a 4G (or even 3G) SIM can be used in a 5G network. However, only a 5G SIM will unlock the full potential of the 5G-SA with the adequate level of security to face the aforementioned challenges.

To support comprehension of the added value of a 5G SIM, TCA has developed a set of recommendations that includes the recent Release 17 support ('Recommended 5G SIM Technical Definition: Enhanced for 3GPP Release 17').

6. Java Card Applet Developer Best Practices

The following section identifies and details a series of best-practices for Java Card applet developers to maximise interoperability.

6.1. Non-Volatile Memory (NVM) Update

Typically, the NVM technology presents a technological limit in the number of times that the memory value can be updated. Typical products may have 200,000 or 500,000 memory cycles, meaning that the same memory area can be updated only 200,000 or 500,000 times without losing its reliability.

To avoid compromising the reliability of the devices, applets are supposed to minimise updates to NVM. This is particularly true if the applet is triggered by relatively frequent events:

- The STATUS command as an example can be received twice a minute, leading to over 1 million triggers in a year. If the applet wrote the same NVM area once per EVENT_STATUS_COMMAND, the UICC reliability could be compromised in less than one year.
- Similarly, the `EVENT_EVENT_DOWNLOAD_LOCATION_STATUS` may also be very frequent, especially when the network signal is unstable or the device is on the border between two different cells.
- Also, the `EVENT_EXTERNAL_FILE_UPDATE` can be very frequent if the file is regularly updated (i.e. the files that have the "Update activity" information set to "High" in the specification where the file is defined, like ETSI TS 102 221 or 3GPP 31.102, such as the EF LOCI [Location Information]).

During the above events, as well as other events that may be very frequent, applets should avoid performing writings in NVM if not necessary, including:

- File update.
- Java Card fields update.
- Java Card persistent array content update.
- Invoking an object constructor.
- Invoking system APIs that may result in NVM update, including the registration or de-registration to toolkit events.

- Invocation of the Garbage Collection (`JCSystem.requestObjectDeletion`); even if no object is deleted it may result in NVM updates.

For cryptographic operations on Java Card 3.0.5 and onward versions, the usage of OneShot class (e.g. `Cipher.OneShot, MessageDigest.OneShot, RandomData.OneShot`) is recommended as it will reduce NVM updates.

```
//For example (Good practice):
.
Cipher.OneShot enc = null;
 try {
     enc = Cipher.OneShot.open(Cipher.CIPHER_RSA, Cipher.PAD_PKCS1);
     enc.init(someRSAKey, Cipher.MODE_ENCRYPT);
     enc.doFinal(someInData, (short) 0, (short) someInData.length,
encData, (short) 0);
 } catch (CryptoException ce) {
     // Handle exception
 } finally {
     if (enc != null) {
         enc.close();
         enc = null;
           }
        }
```

A specific consideration is related to the `javacard.security.KeyPair` object. In fact, the `KeyPair` has two different constructors:

- The `javacard.security.KeyPair(byte algorithm, short keyLength)` constructor, and;
- The `javacard.security.KeyPair(PublicKey publicKey, PrivateKey privateKey)` constructor.

When `javacard.security.KeyPair(byte algorithm, short keyLength)` method is used, the API will instantiate a `KeyPair` object and a pair of `PrivateKey` and `PublicKey` objects. It is not possible in this case to specify if the Key objects use transient or persistent memory.

In some cases, like ephemeral key pairs generation, key usage is limited to the session. This guarantees that the Key objects use volatile memory usage to reduce NVM updates. When the applicative use case requires generation of ephemeral key pairs, the following procedure should be followed:

- For each key type and size to be supported, instantiate only once a `PrivateKey` and a `PublicKey` object using the related `KeyBuilder.buildKey()` methods specifying the usage of transient memory.
- For each key type and size to be supported, instantiate one `KeyPair` using `javacard.security.KeyPair(PublicKey publicKey, PrivateKey privateKey)` constructor and passing the `PrivateKey` and `PublicKey` objects instantiated at previous step.
- Invoke the `KeyPair.genKeyPair()` method each time a generation of new ephemeral keys is needed.

Specific operating systems may have dedicated optimisations to limit the issue, like a wear levelling mechanism (that modifies the physical location to avoid always writing to the same page). But as they are not guaranteed to always be supported by all TCA member eUICC products, applet developers should not rely on such mechanisms to increase robustness.

Initialised arrays declared as static

Arrays containing constants, like menu strings or cryptographic constant arrays, should be declared static. This means the arrays are not initialised at installation, saving code space.

```
//For example (Good practice):

.

.

public final class testApplet extends Applet implements ToolkitInterface
{

       static final byte[] menuEntry = { (byte) 'm', (byte) 'e', (byte)
'n', (byte) 'u', (byte) 'I', (byte) 't', (byte) 'e', (byte) 'm'};

       static final short[] filePath = {(short)0x3F00, (short)0xDF00,
(short)0xEF00};

    .

    .

}
```

Using transient working buffers as scratchpad

Working and scratch buffers should be declared as transient arrays, as its content is supposed to change frequently. If an applet developer needs to manipulate and change data at different offsets within an NVM buffer, they can create a copy of the NVM buffer in a transient buffer. The data manipulation can then be performed on the transient buffer. Finally, the content can be transferred back to the NVM buffer with a single update API invocation, minimising the number of updates of the NVM buffer.

```
//For example (Bad practice)
.
.
public final class testApplet extends Applet implements
ToolkitInterface {
   static final short SIZE_NVM_BUFFER = (short)300;
   static final short NVM_BUFFER_OFFSET_A = (short)0;
   static final short NVM_BUFFER_OFFSET_B = (short)10;
   static final byte NVM_BUFFER_DATA_A = (byte)0x81;
   static final byte NVM_BUFFER_DATA_B = (byte)0x82;

   static byte[] nVM_Buffer = new byte[SIZE_NVM_BUFFER];
   .
   .
   public static void install () {
   new testApplet();
   .
   .
   //Instead of initializing the NVM array directly:
   nVM_Buffer[NVM_BUFFER_OFFSET_A] = NVM_BUFFER_DATA_A;
   nVM_Buffer[NVM_BUFFER_OFFSET_B] = NVM_BUFFER_DATA_B;
   }
}

//For example (Good practice)
.
.
```

```
public final class testApplet extends Applet implements
ToolkitInterface {

   static final short SIZE_NVM_BUFFER = (short)300;

   static final short SIZE_TRANSIENT_BUFFER = (short)30;

   static final short NVM_BUFFER_OFFSET_A = (short)0;

   static final short NVM_BUFFER_OFFSET_B = (short)10;

   static final byte NVM_BUFFER_DATA_A = (byte)0x81;

   static final byte NVM_BUFFER_DATA_B = (byte)0x82;


   static byte[] nVM_Buffer = new byte[SIZE_NVM_BUFFER];

   static byte[] transientBuffer;

   .

   .

   public static void install () {

      transientBuffer = JCSystem.makeTransientByteArray(
SIZE_TRANSIENT_BUFFER, JCSystem.CLEAR_ON_RESET);

      new testApplet();

      .

      .

      //A transient array having sufficient size can be initialized
first

      //and then the nVM_Buffer can be written in one go

      transientBuffer[NVM_BUFFER_OFFSET_A] = NVM_BUFFER_DATA_A;

      transientBuffer[NVM_BUFFER_OFFSET_B] = NVM_BUFFER_DATA_B;

      Util.arrayCopy(transientBuffer, (short)0, nVM_Buffer, (short)0,
SIZE_TRANSIENT_BUFFER);

   }

   .

   .

}
```

Using a single contiguous array to hold multiple arrays of the same type

Instead of creating several arrays of the same data type, it's better to create one for each data type and define constants as offsets to access the desired zone within the array. Since arrays are objects, their creation is slow and consumes more memory as the virtual machine adds some more bytes as header.

```
//For example (Bad practice)

.

.

public final class testApplet extends Applet implements ToolkitInterface {

   private static ToolkitRegistry tlkReg;

   private static byte menuID1;

   private static byte menuID2;

   private static byte menuID3;

   .

   //Instead of creating several byte arrays like below

   static final byte[] menuEntry1 = {(byte) 'm', (byte) 'e', (byte) 'n',
(byte) 'u', (byte) 'I', (byte) 't', (byte) 'e', (byte) 'm', (byte) '1'};

   static final byte[] menuEntry2 = {(byte) 'm', (byte) 'e', (byte) 'n',
(byte) 'u', (byte) 'I', (byte) 't', (byte) 'e', (byte) 'm', (byte) '2'};

   static final byte[] menuEntry3 = {(byte) 'm', (byte) 'e', (byte) 'n',
(byte) 'u', (byte) 'I', (byte) 't', (byte) 'e', (byte) 'm', (byte) '3'};

   .

   .

   public static void install () {

      new testApplet.register();

      tlkReg = ToolkitRegistrySystem.getEntry();

      menuID1 = tlkReg.initMenuEntry(menuEntry1, (short)0, (short)
menuEntry1.length, (byte) 0, false, (byte) 0, (byte) 0);

      menuID2 = tlkReg.initMenuEntry(menuEntry2, (short)0, (short)
menuEntry2.length, (byte) 0, false, (byte) 0, (byte) 0);

      menuID3 = tlkReg.initMenuEntry(menuEntry3, (short)0, (short)
menuEntry3.length, (byte) 0, false, (byte) 0, (byte) 0);

      .

      .

   }

   .
   .
}


//For example (Good practice)
public final class testApplet extends Applet implements ToolkitInterface {
   private static ToolkitRegistry tlkReg;


   //Create a single array that can hold all the data of the same type
   private static byte[] menuID = new byte[3];
   private static final byte[] constantByteData = {
      (byte) 'm', (byte) 'e', (byte) 'n', (byte) 'u', (byte) 'I', (byte)
't', (byte) 'e', (byte) 'm', (byte) '1',
      (byte) 'm', (byte) 'e', (byte) 'n', (byte) 'u', (byte) 'I', (byte)
't', (byte) 'e', (byte) 'm', (byte) '2',
      (byte) 'm', (byte) 'e', (byte) 'n', (byte) 'u', (byte) 'I', (byte)
't', (byte) 'e', (byte) 'm', (byte) '3'};
   private static final short OFFSET_MENU_ITEM1 = 0;
   private static final short OFFSET_MENU_ITEM2 = 9;
   private static final short OFFSET_MENU_ITEM3 = 18;
   .
   .
   public static void install () {
      new testApplet.register();
      tlkReg = ToolkitRegistrySystem.getEntry();
      menuID[0] = tlkReg.initMenuEntry(constantByteData, OFFSET_MENU_ITEM1,
(short) 9, (byte) 0, false, (byte) 0, (byte) 0);
      menuID[1] = tlkReg.initMenuEntry(constantByteData, OFFSET_MENU_ITEM2,
(short) 9, (byte) 0, false, (byte) 0, (byte) 0);
      menuID[2] = tlkReg.initMenuEntry(constantByteData, OFFSET_MENU_ITEM3,
(short) 9, (byte) 0, false, (byte) 0, (byte) 0);
      .
      .
   }
   .
   .
}
```

6.2. Fields and Local Variables

As field variables are stored in NVM, frequent updates should be avoided to reduce stress on the NVM. If a field variable is subject to change due to some calculations, it is advised to perform the calculation in a local variable and assign the result to the field variable.

If a field member (primitive or references to objects/arrays) is accessed several times in the same method, copy the content of the field in a local variable and use the local variable (reducing the access to the field member).

Due to the size of the stack, methods should avoid using too many local variables and parameters. Whenever possible it is suggested to reuse local variables and limit the scope of the local variables to be only be accessible within a block.

6.3. Applet Deletion

If an object owned by an applet instance is used by another applet instance, the object must be dereferenced before deleting the owning applet instance. The uninstall method can be used to remove all dependencies before applet deletion.

Static variables and methods belong to the class, not to the instance, so they are not affected by the firewall and are accessible by other applet instances. However, caution must be exercised when using static methods and fields. If the CAP file containing the static methods or fields is to be deleted, the CAP file using the method should be removed first. CAP files, whose static methods or fields are used by another CAP file, will not be deleted unless those CAP files are deleted first.

6.4. Object Constructors

As much as possible, object (for keys, PIN, signature etc.) should be created in the applet's constructor. When created during runtime, it is not guaranteed that there will be enough memory.

Byte array initialisation should also be done in the applet constructor.

Below is a non-exhaustive list of the methods which create new object when invoked:

- `getInstance`
- `makeTransientByteArray`
- `makeTransientBooleanArray`
- `makeTransientObjectArray`
- `makeTransientShortArray`
- `buildKey`

- `UICCSystem.getTheUICCView`
- `buildTLVHandler`

It is recommended to use Java Card RE owned exception objects instead of creating new objects for these exceptions (e.g. `ISOException`). Developers can use `javacard.framework.UserException` for custom exceptions.

6.5. Stack Management

The more a method contains local variables and parameters, the more stack it will consume. As the RAM size is limited, the developer should refrain from using many local variable and parameters in a method.

Deeply nested method calls (method calling another method which in turn call another method, etc.) and use of recursion should be avoided.

```
// Wrong practice
private void method(){
      byte tmpByte = methodA();
      ..
      ..
}



private byte methodA(){
    byte l_byte
      ...
      ...
      l_byte = methodB(l_byte);
      return l_byte;
}



private byte methodB(byte param){
      ...
      ...
      return param;
}
```

```
// Good practice
private void method (){
       byte tmpByte = methodA();
       tmpByte = methodB(tmpByte)
}



private byte methodA (){
     byte l_byte
       ...
       ...
       return l_byte;
}



private byte methodB (byte param){
       ...
       ...
       return param;
}
```

6.6. Handlers (Re-entrance)

When retrieving the `ProactiveHandler`, the `getTheHandler` should be surrounded by a try-catch block. In case the handler is not available, a flag can be set to indicate this state.

Whether the proactive command is needed, the flag can be checked for availability. The event `EVENT_PROACTIVE_HANDLER_AVAILABLE` can be registered for re-entrance if needed. As event registration results in a NVM update, however, it should be avoided when not required.

If the event `EVENT_PROACTIVE_HANDLER_AVAILABLE` is expected to be utilised frequently, the event `EVENT_STATUS_COMMAND` can serve as an alternative in order to reduce NVM wearing.

```
//For example (Good practice)

public final class testApplet extends Applet implements

ToolkitInterface, ToolkitConstants {

   private static ToolkitRegistry tlkReg;


   private static final short TRANS_OFF_TASK = 0;

   private static final short TRANS_LEN_TASK = 1;


   private static final short TRANS_SIZE = TRANS_OFF_TASK

            + TRANS_LEN_TASK;

   private static final byte TASK_SEND_SMS = 1;

   private byte[] transAry;

   .

   .

   public static void install (byte bArray[], short bOffset, byte
bLength) {

      new testApplet.register(bArray,

        (short)(bOffset + 1),

        (byte)bArray[bOffset]);

      tlkReg = ToolkitRegistrySystem.getEntry();

      tlkReg.setEvent(EVENT_EVENT_DOWNLOAD_LOCATION_STATUS);


        transAry = JCSystem.makeTransientByteArray(TRANS_SIZE,

        JCSystem.CLEAR_ON_RESET);

   }

   .

   .

   public void processToolkit(short sEvent) {

         // variable to check whenever the proactive command handler is
needed

         boolean isHandlerAvailable = false;

      try{

            ProactiveHandlerSystem.getTheHandler();

                  isHandlerAvailable = true;
```

```
          }catch(ToolkitException tlkEx){
             // handler not available
          }

       switch (sEvent) {
       case EVENT_EVENT_DOWNLOAD_LOCATION_STATUS:

          doSomeThingWithEnvelop();
          if (isHandlerAvailable){
                   sendSMS();
          }else{
             tlkReg.setEvent(EVENT_PROACTIVE_HANDLER_AVAILABLE);
             transAry[TRANS_OFF_TASK] = TASK_SEND_SMS;
          }
       break;

       case EVENT_PROACTIVE_HANDLER_AVAILABLE:
          switch(transAry[TRANS_OFF_TASK]){
                   case TASK_SEND_SMS:
                        sendSMS();
                   break;

              }
              // clear flag
              transAry[TRANS_OFF_TASK] = 0;
       break;

       default:
       break;
       }
    }

    private void sendSMS(){
          ProactiveHandler pro =
            ProactiveHandlerSystem.getTheHandler();
            ...
            ...
    }

}
```

6.7. Execution Time

Applet developers should limit the execution time of specific commands to specified time limits to avoid blocking the device in case it requires UICC services. Typically, the execution time of a single command should be below 30 seconds. As the execution time depends on the target platform (the JCVM on some chips can be faster than others), there can be significant differences between devices provided by different vendors. Specific operations in particular (like asymmetric cryptography, NVM buffer initialisations, data manipulation in arrays, etc.) may take a significant amount of time. It is then advised to include some margins within the execution time to increase confidence that the execution time is always under 30 seconds. In case a command is suspected to require more than 30 seconds, it is advised to introduce MORE TIME proactive command to interrupt the execution and give back control to the mobile phone and continue the execution at the corresponding TERMINAL RESPONSE command.

```
//For example (Bad practice)
public final class testApplet extends Applet implements
ToolkitInterface, ToolkitConstants {
    static ToolkitRegistry tlkReg;
    static final short SIZE_NVM_BUFFER = (short) 30000;
    static boolean initializeLargeNVMBuffer = true;
    static byte[] nVM_Buffer = new byte[SIZE_NVM_BUFFER];
    .
    .
    public static void install () {
        new testApplet().register();
        tlkReg = ToolkitRegistrySystem.getEntry();
        tlkReg.setEvent(EVENT_PROFILE_DOWNLOAD);
    }
```

```
   .

   .

   public void processToolkit(short sEvent) {

      switch(sEvent) {

      case EVENT_PROFILE_DOWNLOAD:

         if(initializeLargeNVMBuffer) {

            Util.arrayFill(nVM_Buffer, (short) 0, SIZE_NVM_BUFFER, (byte)
0xFF);

            initializeLargeNVMBuffer = false;

         }

      }

   }

}


//For example (Good practice)

public final class testApplet extends Applet implements ToolkitInterface,
ToolkitConstants {

   static ToolkitRegistry tlkReg;

   static final short SIZE_NVM_BUFFER = (short) 30000;

   static final short SIZE_NVM_BUFFER_BLOCK = (short) 10000;

   static final short SIZE_TRANSIENT_BUFFER = 2;

   static final short OFFSET_TBUFF_NBUFF_BLOCK = 0;

   static final short OFFSET_TBUFF_HDLR_AVL = 1;

   static final byte MASK_INIT_NBUFF = 0x01;

   static final byte NVM_BUFFER_BLOCKS = (byte) ( SIZE_NVM_BUFFER /
SIZE_NVM_BUFFER_BLOCK);


   static boolean initializeLargeNVMBuffer = true;


   static byte[] nVM_Buffer = new byte[SIZE_NVM_BUFFER];

   static byte[] transientBuffer;

   .

   .

   public static void install () {

      transientBuffer = JCSystem.makeTransientByteArray((short)
SIZE_TRANSIENT_BUFFER, JCSystem.CLEAR_ON_RESET);
```

```
      new testApplet().register();

      tlkReg = ToolkitRegistrySystem.getEntry();

      tlkReg.setEvent(EVENT_PROFILE_DOWNLOAD);

   }

   .

   .

   public void processToolkit(short sEvent) {

      switch(sEvent) {

         case EVENT_PROFILE_DOWNLOAD:

            initializeLargeNVMBuffer();

         break;


         case EVENT_PROACTIVE_HANDLER_AVAILABLE:

            if((transientBuffer[OFFSET_TBUFF_HDLR_AVL] & MASK_INIT_NBUFF)
== MASK_INIT_NBUFF) {

               transientBuffer[OFFSET_TBUFF_HDLR_AVL] &=
~MASK_INIT_NBUFF;

               initializeLargeNVMBuffer();

            }

         break;

         .

         .

         default:

         break;

      }

   }


   private boolean sendMoreTime(byte mask) {

      try {

         ProactiveHandler proHdlr =
ProactiveHandlerSystem.getTheHandler();

         proHdlr.initMoreTime();

         proHdlr.send();

         return true;

      }catch (ToolkitException e) {
         transientBuffer[OFFSET_TBUFF_HDLR_AVL] |= mask;
         tlkReg.setEvent(EVENT_PROACTIVE_HANDLER_AVAILABLE);
      }
      return false;
   }

   private void initializeLargeNVMBuffer()
   {
      byte nVM_Block = transientBuffer[OFFSET_TBUFF_NBUFF_BLOCK];

      if(!initializeLargeNVMBuffer)
         return;

      while(nVM_Block < NVM_BUFFER_BLOCKS) {
         if(!sendMoreTime(MASK_INIT_NBUFF))
            return;
         Util.arrayFill(nVM_Buffer, (short) (nVM_Block *
SIZE_NVM_BUFFER_BLOCK), SIZE_NVM_BUFFER_BLOCK, (byte) 0xFF);
         transientBuffer[OFFSET_TBUFF_NBUFF_BLOCK]++;
         nVM_Block++;
      }
      initializeLargeNVMBuffer = false;
   }
}
```

6.8. RAM Management

Applets should limit the usage of RAM to the minimum, as the available resources in the targeted eUICC where the profile containing the applet will be loaded is not known.

An efficient way to save RAM is reusing the same scratch buffers or using system allocated buffers when available (like the APDU buffer or the toolkit volatile byte array). In particular, the APDU buffer is only available when the access is performed by the currently selected applet, while the toolkit volatile byte array is only available when the buffer reference is retrieved by the currently selected applet or by the applet currently triggered by a toolkit event.

6.9. Object Creation within Transactions

Applet developers should be aware of the following requirement from the section 7.6.3 Cleanup Responsibilities of the Java Card RE specification [Java Card 3.0.5] –

"Programmatic abortion after creating objects within the transaction can be deemed to be a programming error. When this occurs, the Java Card RE may, to ensure the security of the card and to avoid heap space loss, lock up the card session to force tear or reset processing."

This means that applet developers should avoid creating objects within transactions to avoid interoperability issues on platforms which *force tear or reset processing*.

6.10. Shareable Interface and Multi-Selection

It is recommended that the server applets providing access via shareable interfaces also implement the `MultiSelectable` interface. This is to avoid the `SecurityException` being thrown by Java Card RE when the shareable interface is requested/accessed while the context of the server applet instance is already active on another logical channel or I/O interface (as described in section 6.2 of Java Card RE specification and Java Card API specification).

In addition, some standard APIs (e.g. ETSI 102 267 `uicc.connection` , ETSI 102 241 `uicc.suspendresume`) define interfaces which are not extending `javacard.framework.Shareable`.

To ensure interoperability TCA members recommend that applet developers implement the `MultiSelectable` interface in their applets if such APIs are used.

Note: On specific products in the field from TCA members, an applet can also be triggered by such APIs when its context is already selected on a different channel even if it does not implement the `MultiSelectable` interface. For such applets already in the field, it is recommended to add `MultiSelectable` support whenever an applet modification is applied to increase interoperability.

6.11. CLEAR_ON_DESELECT Memory Access

Applets shall avoid use of `CLEAR_ON_DESELECT` memory within the shareable interface APIs, wherever these APIs are not executing in the currently selected applet context.

6.12. Exception Handling

Applets should handle all exceptions arising out of a Java Card/GlobalPlatform/ETSI API invocation to be able to continue processing depending on the business logic (e.g.

`ProactiveHandler` availability should be ensured before invoking the related APIs or properly handle the related `HANDLER_NOT_AVAILABLE` exception).

6.13. Transactions

The number of bytes of conditional updates within a transaction depends on the capacity of the commit buffer. The following aspects of the transaction are implementation dependent and may vary across card manufacturers:

- The capacity of the commit buffer.
- The number of bytes consumed in the commit buffer for each conditional update. It may be more than the actual number of bytes written by the applet logic.

A `TransactionException` is thrown if the conditional update results in exceeding the commit capacity. Care should be taken by the applet developers to:

- Implement appropriate logic to handle transactions that may exceed this commit capacity and;
- Abort the transaction as part of the exception handling logic (within the try-catch-finally block).

Note: Nested transactions shall be avoided as per Java Card specification.

6.14. CAP File Generation

Only standard converters shall be used to generate applet CAP files. Presence of proprietary bytecodes within the CAP content will affect the interoperability and shall be avoided.

6.15. Toolkit Install Parameters

Applet developers should be aware of the restrictions specified in section 8.2.1.3.2 of [ETSI 102 226] and section 21.2.3 of [Interoperability Stepping Stones R7] which state that:

- Applets implementing the `uicc.toolkit.ToolkitInterface` interface or `uicc.access.FileView` interface shall be installed with the *UICC System Specific Parameters* (*Tag* *'EA'*) TLV object and those implementing the `sim.toolkit.ToolkitInterface` interface or using the `sim.access.SIMView` interface shall only be installed the *SIM File Access and Toolkit Application Specific* Parameters *(Tag 'CA')* TLV object.

It is recommended that applet developers avoid use of the `sim.toolkit.ToolkitInterface` interface and the `sim.access.SIMView` interface since the eUICC is not mandated to support 2G SIM applications and instead switch to the usage of the `uicc.toolkit.ToolkitInterface` interface and the `uicc.access.FileView` interface.

6.16. FileView API Usage

As defined in section 12.8 of [Stepping Stones R7], applet developers shall ensure that applets using the `FileView` and `SIMView` APIs avoid using them interchangeably (e.g. using one API type for file selection and a different one for read/update, i.e. select using `FileView` API and read/update using `SIMView` API).

6.17. Menu Entry Initialisation

Toolkit applets shall ensure that the Menu Entries are initialised correctly (during the installation as described in ETSI 102 241 – `initMenuEntry()` method of `ToolkitRegistry` interface). Otherwise the menu entry state, returned as part of the response to a GET STATUS command for retrieving the Menu Parameters, may be undefined.

6.18. Optimised API Usage

Applets should make use of the right APIs to avoid unnecessary processing overhead in the eUICCs, which is not intended as per the business use-case.

- Use file selection API which does not require the FCP generation when the intention is to just select the file and perform some operation on it (either read, write, activate, deactivate etc.).
- Avoid passing NVM buffers as arguments when not intended.
- Avoid invoking read/update APIs which do nothing (i.e. passed with length argument as 0).

6.19. SUCI API

Typically, SUCI calculation is performed by the USIM application (underlying OS) when the services n°124 and n°125 are indicated as available in the EF_{UST} of the USIM NAA. But there may be use-cases where a mobile operator prefers that this operation be performed by a dedicated applet in the profile.

Such an applet implements the `SUCICalculator` interface, and it should ensure/be aware of the following:

- Even though the 3GPP specification [3GPP 31 102] allows the use of compressed elliptic curve public keys for the ProfileB scheme, the home network public key is provisioned in uncompressed format according to the TCA specification [TCA eUICC Profile Package: IFTS v3.3].

- Storing sensitive data in the buffer passed to the `getSUCI()` API is avoided since this buffer is a global array.
- The length of the SUCI output is strictly within the limit specified by the “bLength” parameter of the `getSUCI()` API.
- The ephemeral keys are regenerated for every call to the `getSUCI()` API.
- The sensitive data is handled with appropriate security measures and is erased securely immediately after use (including exception scenarios).

The profile creator shall ensure that the SUCI applet is granted the required access rights (in the Access Domain field of the install parameters) to access the files required for the SUCI calculation.

7. Java Card Applet Development Guidelines for Secure Products

In the 3GPP telecom architecture, the UICC – as a tamper-resistant secure hardware component – contains the most sensitive information on the user equipment (UE) side, namely the storage and processing of the subscription credentials. In order to maintain the highest level of security, it is important for the applets to be properly developed. This is even more important with the evolution to eUICC which are securely certified and where several profiles from different MNOs can be present at the same time.

There are two types of applets: basic and sensitive. Basic applets don't have assets to protect, nevertheless they have to comply to a minimum of security rules, especially for those installed on eUICC products (as defined in GSMA SGP.05 and SGP.25 and re-highlighted hereafter). Sensitive applets are applets which store or manipulate their own security assets. The following recommendations outline how the applet developer can sufficiently protect the applet assets.

7.1. Security Recommendations for All Applets

- Generic Rules:
  - Java Card applets must, at a minimum, follow the "GlobalPlatform Card Composition Model Security Guidelines for Basic Applications". In particular, Java Card applets must successfully pass byte code verification using tools from Oracle. The tools used for byte code verifications shall be the latest versions available.
  - Java Card applet AID must be set as defined in ETSI 101220. In particular, applet developer shall use its own RID registered at ISO as defined in 7816-5.

- Standard APIs:
  - The standard API should be used whenever possible, rather than rewriting methods. This holds for:
    - Java Card standard API
    - GlobalPlatform API
    - UICC API
    - USIM API

7.2. Additional Security Recommendations for Sensitive Applets

- Sensitive Data Management:
    - Sensitive data must be initialised at the beginning and cleared at the end of the session.
    - Sensitive data should be stored in transient data.
    - Always clear, with random data, (global) arrays used to store temporarily sensitive data.
    - Confidential data must not be stored in plain (e.g. may be ciphered or masked and stored).
    - Sensitive constant value: When a constant is used as reference value for a sensitive action, avoid choosing 0x00 or 0xFF for this constant.

- Rollback Attacks
    - Protect your sensitive data against rollback attacks.

- Flow Control:
    - In order to protect against multiple perturbations, countermeasures should be implemented to detect any change to the normal execution flow. If an inconsistent state is reached, an appropriate measure shall be applied according to applicable context (e.g. block the application, reset).

- Sensitive Standard API:
    - When using a method of a standard API that needs absolutely to be executed, some consistency checking must be done to assume it has been correctly executed. Since Java Card 3.0.5, the SensitiveResult class can be used for asserting results of sensitive functions.

- Random:
    - Avoid using deprecated random (`ALG_PSEUDO_RANDOM` and `ALG_SECURE_RANDOM`). Always use appropriate random depending on the usage. In particular, always use either `ALG_KEYGENERATION` or `ALG_TRNG` algorithms for sensitive use cases.

- Programmatic Exceptions
  - Avoid the usage of programmatic exceptions to exit from a loop (e.g. do not parse table until catching an index out of bounds exception).

- Java Card RMI:
  - The usage of the Java Card RMI mechanism is prohibited, because it lacks security-related features (e.g. authentication and secure channels)

8. Interoperability Testing Tools, Services and Events

In the context of OS and applets, the ability of applets to run and function properly on different OS's without requiring modification or adaptation for each distinct OS is referred to as interoperability.

The eSIM ecosystem is enabled by an established infrastructure and global specifications, and all individual eSIM products and components are tested extensively prior to deployment. Due to variations in the interpretation and implementation of industry specifications, however, some interoperability issues only emerge when solutions are deployed live in the field and interact with other participants and components across the ecosystem.

As the eSIM market grows in size and complexity, so too does the risk and impact of interoperability issues. A key challenge is the insufficient and incomplete testing of eSIM profiles, which leads to compatibility issues with specific devices on the market. The interoperability is even more challenging when the eSIM profile contains a Java Card applet.

To help Java Card developers ensure seamless integrations, this section summarises key interoperability testing tool, services and events. This includes the TCA Loader, TCA eSIM Interoperability Service, and insights into interoperability events, such as the GlobalPlatform Interoperability Test Fest.

8.1. TCA Loader

The TCA Loader is a free to use tool to promote the interoperable deployment of SIM-based value-added services.

Version 3.0 of the TCA Loader provides applet developers with the ability to perform complete applet management on Java Card smartcards. This includes exploring the card contents and downloading, installing and deleting applications. The TCA Loader supports various security protocols such as SCP02, SCP03, SCP80, SCP81 and CAT-TP.

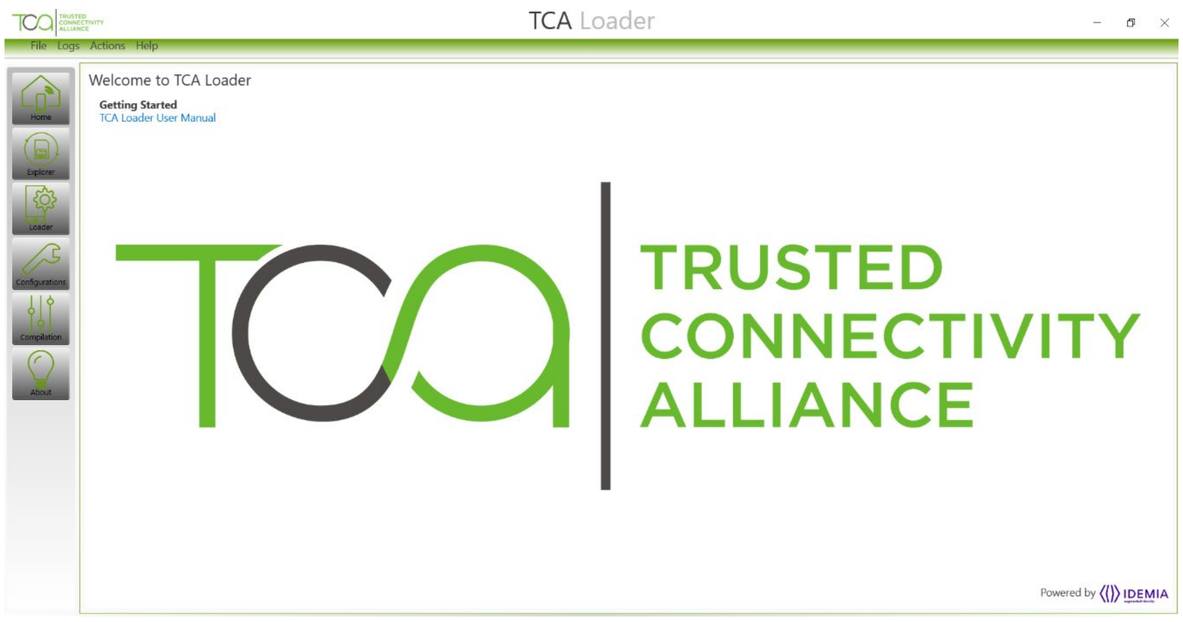

Figure 1 - TCA Loader Main Page

The TCA Loader also provides applet developers with a convenient user interface for applet compilation. The applet developer can choose the library classpath, export file path, then store the compilation configuration. Applet developers can then import the stored configuration later on.

The TCA Loader can also be used to simulate the actual card behaviour in the field when performing application management via OTA or I/O using various supported security protocols. This enables applet developers to ensure applet management functionalities prior to card deployment.

Downloading and Using the TCA Loader

The TCA Loader can be downloaded here from the TCA website:

- Direct Link

- Summary Page

The Full TCA Loader user manual is available after installation on Home > Getting Started > TCA Loader Manual, or About > User Manual.

8.2. TCA eSIM Interoperability Service

To address eSIM profile interoperability issues, TCA has launched the TCA eSIM Interoperability Service – delivered by COMPRION . The service enables operators to test how their eSIM profiles, which may contain Java Card applet, interact with an extensive range of consumer eSIM devices.

Service Description

Test one (or more) customer-provided eSIM profile(s) against a customer-defined subset of the available portfolio of test devices.

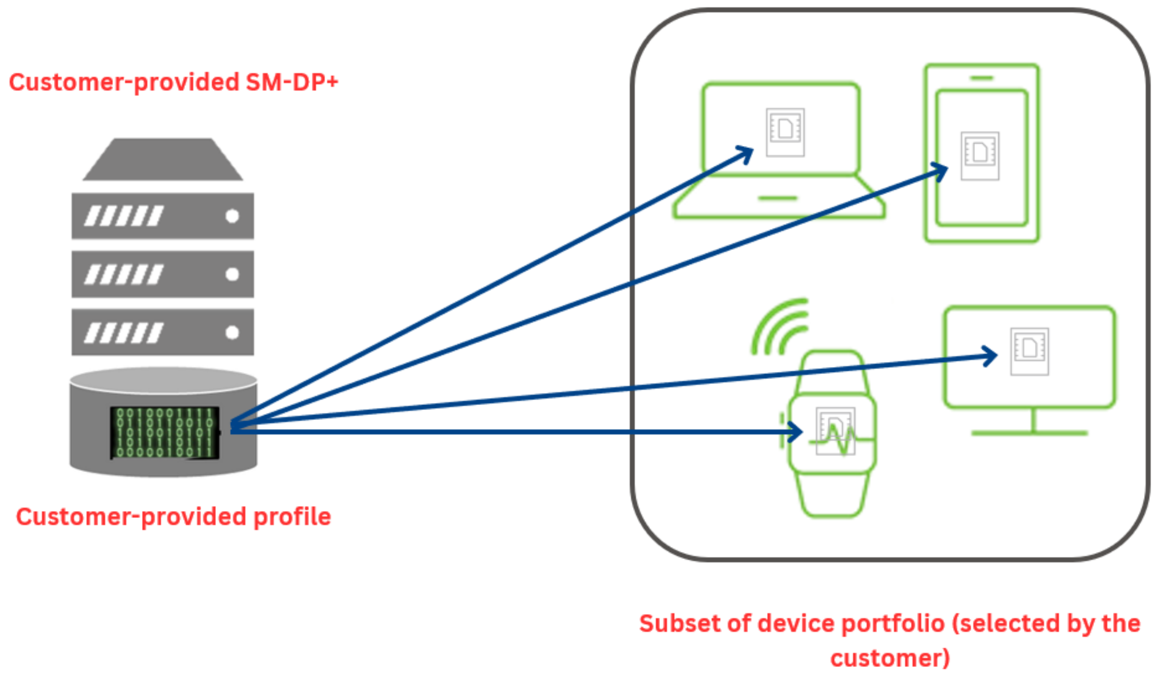
Customer-provided SM-DP+
Customer-provided profile
Subset of device portfolio (selected by the customer)

Figure 2 - Testing profile against test devices

The testing is conducted between SM-DP+ services, using the GSMA live certificates, and devices with eSIMs also using GSMA live certificates.

To commence testing, the customer ensures that the eSIM profile(s) can be downloaded from a customer-provided SM-DP+ service by scanning QR codes or entering activation codes. There are three possible ways to conduct the testing:

- **Self-testing:** the testing is done by the customer on COMPRION's site in Paderborn, Germany.

- **Self-testing remotely**: the testing is done by the customer remotely. When technically possible, COMPRION enables the customer to remotely access the test devices.
- **Testing-by-COMPRION:** the testing is done solely by COMPRION's expert

Testing Process

The tester selects one of the devices from the test device portfolio and initiates a profile download. Once the profile is successfully downloaded, a number of scenarios can be tested, including:

- Initiating a call

- Receiving a call

- Sending an SMS

- Receiving an SMS

- Setting up internet connection

Besides these basic checks the customer can request to add further checks (e.g. Access Point Name [APN] verification).

If the profile contains a Java Card applet, the customer can also ask to test the behaviour of the applet. The customer needs to describe the behaviour which can be detected on the user interface and the potential interactions between the user and the applet. Based on this description, the behaviour of the applet can be tested.

The behaviour of the device is then documented. In case any interoperability issues are detected, they are described in detail together with print screens and a video recording (if requested). The final documentation is then shared with the customer.

In case the issue cannot be resolved by the customer, COMPRION offers root cause analysis as part of the Advanced eSIM Interoperability Testing Service.

While the current scope of the TCA eSIM Interoperability Service is SGP.22, it can later be extended to SGP.32.

Using the TCA eSIM Interoperability Service

For more information about the service, please visit the TCA and COMPRION websites:

- TCA eSIM Interoperability Service: Product Page
- Five Step Guide to Using the TCA eSIM Interoperability Service
- TCA eSIM Interoperability Service: Product Sheet

- COMPRION – TCA eSIM Interoperability Service

8.3. GlobalPlatform Test Fest

'Test Fest' means an event conducted by GlobalPlatform for the purpose of enabling product vendors to engage in cross-testing of products, in order to demonstrate compliance with the relevant specifications from GlobalPlatform or other SDOs (like GSMA or TCA).

The Test Fest environment also offers participants the ability to interact with one another for purposes of improving their individual products and providing feedback on the specifications/configurations themselves.

The purpose of the Test Fest is to validate:

- The specification compliance test suite:
    - Suite consisting of testing documentation, test scripts and/or other materials, based on a given test specification and related configuration, which has been released by GlobalPlatform or other SDOs for purposes of enabling authorized users to develop corresponding qualified test products.

- The test tools implementing the GlobalPlatform test suite
    - Tools that integrate any portion of the test suite and are created, developed or produced for the purpose of performing tests on proposed compliant products.

8.4. GSMA LITE Event

The GSMA LITE Event is GSMA's Live Interoperability Test Event, which has been held three times.

The events are attended by companies ranging from MNOs/MVNOs to eUICC manufacturers, device OEMs and test companies, enabling various eSIM profiles to be tested against multiple eUICCs to achieve interoperability.

9. Interoperability Checklists

Interoperability Checklist for Applet Developers

Title	Description	Check
1. No object creation within a transaction	Ensure that objects are not created within a transaction	☐
2. CLEAR_ON_DESELECT memory access	Ensure that `CLEAR_ON_DESELECT` memory is not accessed within a shareable interface APIs that could be executed in a context different from the context of the currently selected applet	☐
3. Exception handling	Ensure that the exceptions thrown from APIs are caught and handled accordingly by use of try-catch block	☐
4. Transactions capacity	Ensure that the number of conditional updates within a transaction is within the capacity of the commit buffer	☐
5. Transactions exception handling	Ensure that the exceptions thrown within a transaction are caught and handled accordingly by use of try-catch block.	☐
6. Aborting transaction	Ensure that the transaction is programmatically aborted in the exception handling logic in case of any error within a transaction	☐
7. CAP file generation	Avoid proprietary byte codes in the CAP file. Ensure that standard converters are used to generate CAP files	☐
8. Usage of `sim.toolkit.ToolkitInterface` interface and the `sim.access.SIMView` interface	Avoid use of the `sim.toolkit.ToolkitInterface` interface and the `sim.access.SIMView` interface since the eUICC is not mandated to support 2G SIM applications and instead use the `uicc.toolkit.ToolkitInterface` interface and the `uicc.access.FileView interface`.	☐
9. `FileView` and `SIMView` APIs	Ensure that the `FileView` and `SIMView` APIs are not used interchangeably within the same applet	☐
10. Menu Entry initialisation	Ensure that the Menu Entries (if used within the applet) are initialised correctly during the installation.	☐

TRUSTED CONNECTIVITY ALLIANCE

11. Optimised API usage	Avoid unnecessary processing overhead by - Using the right APIs (wherever there are multiple APIs available to perform similar operation) - Avoiding the use of NVM buffer - Passing the right value for the arguments in the API	☐
12. SUCI API – Sensitive data	Ensure that sensitive data is • not stored in the buffer passed to the `getSUCI()` API • handled with appropriate security measure and is erased securely immediately after use (including exception scenarios).	☐
13. SUCI API – Ephemeral keys	Ensure that the ephemeral keys are regenerated for every call to the `getSUCI()` API	☐
14. SUCI API – SUCI output	Ensure that the length of the SUCI output is strictly within the limit specified by the “`bLength`” parameter of the `getSUCI()` API	☐
15. NVM update	Limit the updates to the same NVM (Flash or EEP) variables and arrays to less than 200,000 for the lifecycle of the applet	☐
16. NVM update on STATUS	Avoid NVM update on `EVENT_STATUS_COMMAND` event	☐
17. NVM update on Location Status	Avoid NVM update on `EVENT_EVENT_DOWNLOAD_LOCATION_STATUS` event	☐
18. NVM update on File Update Event	Avoid NVM update on File Update Events for files with “High Update activity”	☐
19. Limit NVM update operations	Limit all the operations that may cause NVM operations to the strict minimum, including: - File update - Java Card fields update - Java Card persistent array content update - Invoking an object constructor	☐

	- Invoking System APIs that may result in NVM update, including: o the registration or de-registration to toolkit events o calling select methods of the `uicc.access.FileView` interface that is created as "`NOT_A_TRANSIENT_OBJECT`"	
20. Use OneShot class	If the target device supports Java Card v3.0.5, use the `OneShot` class for cryptographic operations	☐
21. Execution time	Verify that every toolkit execution is limited to 30 seconds under any circumstances. If an execution takes more time interrupt it with the MORE TIME proactive command	☐
22. RAM usage	Minimise RAM allocation to the minimum by reusing the same scratch buffers or using system allocated buffers when available (like the APDU buffer or the toolkit volatile byte array)	☐
23. Object creation	Avoid invoking object constructor or factory method on non-installation or non-personalisation code	☐
24. Data integrity	- Verify that the interdependent fields are updated in atomic transaction - Verify that the update operation of sensitive array or file is done in atomic way	☐
25. API usage	Verify that the applet does not use proprietary library	☐
26. Converter usage	- Verify that the applet is generated using standard Oracle converter - Verify that the applet has been verified by Oracle off-card verifier	☐
27. Bytecode verification	- Verify that the applet passes the latest bytecode verification process by Oracle	☐
28. Java Card RMI	- Do not use Java Card RMI	☐
29. Sensitive data management	Only applicable to sensitive applet - all rules on sensitive data implemented	☐

30. Rollback protection	Only applicable to sensitive applet - rollback protected	☐
31. Flow control	Only applicable to sensitive applet - flow control implemented	☐
32. Sensitive standard API	Only applicable to sensitive applet - Use `SensitiveResult` class when possible	☐
33. Random	Only applicable to sensitive applet - Use `ALG_KEYGENERATION` or `ALG_TRNG`	☐
34. Shareable interface and multi-selection	It is recommended that the server applets providing access via shareable interfaces also implement the `MultiSelectable` interface for use cases where the shareable interface is requested/accessed while the context of the server applet instance is already active on another logical channel or I/O interface	☐

Interoperability Checklist for Applet Installation and Configuration		
Title	**Description**	**Check**
1. Toolkit installation parameter	Ensure that the applets - implementing the `uicc.toolkit.ToolkitInterface` interface or using the `uicc.access.FileView` interface are installed with the *UICC System Specific Parameters* (Tag ‘EA’) TLV object - implementing the `sim.toolkit.ToolkitInterface` interface or using the `sim.access.SIMView` interface are installed with the *SIM File Access and Toolkit Application Specific Parameters* (Tag ‘CA’) TLV object	☐
2. SUCI API – key format	Ensure that for the ProfileB scheme, the home network public key is provisioned in uncompressed format	☐

3. SUCI calculator applet - installation parameter	Ensure that the applet implementing the `SUCICalculator` interface is granted with the required access rights (in the Access Domain field of the install parameters).	☐

10. About Trusted Connectivity Alliance

Trusted Connectivity Alliance (TCA) is a global industry association working to enable trust in a connected future.

The organisation evolved from the SIMalliance, reflecting the continued expansion of the global SIM industry and the need for broader collaboration. Its members are leading providers of secure connectivity solutions for consumer, IoT and M2M devices. This spans Tamper Resistant Element (TRE) technologies including SIM, eSIM, integrated SIM, embedded Secure Element (eSE) and integrated Secure Element (iSE), as well as hardware and software provisioning and other personalisation services.

TCA members are: Card Centric, COMPRION, Eastcompeace, Giesecke+Devrient, IDEMIA, Kigen, Linxens, Monty Mobile, NXP Semiconductors, Oasis Smart SIM, STMicroelectronics, Thales, Valid, Workz Group, Wuhan Tianyu and XH Smart Card.

www.trustedconnectivityalliance.org | News | Blog | X | LinkedIn | YouTube