Java Card™ & STK Applet Development Guidelines

Version 1.0

Table of content

1 Introduction

1.1 Purpose

The aim of this document is to provide guidance in the development of your java card applet. The very mandatory and basic things to be known and done for such a development are described below, and you will find a checklist at the end of this document that you can use for each of your projects.

The document also aims at describing a Java Card™ code sensitive to card tear-out and at providing related development guidelines. Not taking those rules into account may lead to severe damage since the cards on the field may become out of order with according consequences for customers and the company

This document focuses on Java Card especially and assumes the reader knows basic Java programming thus will not be covering Java programming syntax and coding techniques.

1.2 BEWARE

The Java and Java Card memory models are **different** and are therefore requesting different ways of design.

EEPROM is stressed each time you use it and the number of time you can update an EEPROM cell is limited (guaranteed 100,000 updates for E2PROM and FLASH technologies).

In the meantime, **RAM space is very limited in Java Card** memory models compared to Java memory model.

2 Applet Development

2.1 Process for Applet Deployment

Value added services are implemented on the Java card in the form of applets.

The applets have to be developed by the application developers and load into the card using the appropriate commands. When you need to develop an applet you must remember:

- To use the classes defined in TS 43.019, Java Card™ 2.1.1
- To use only the subset of the Java™ language defined in the Java Card™ 2.1.1 standards.

The process from service definition to application deployment is briefly described in the following parts.

2.1.1 Service definition

The services are defined by the mobile communication operator or other service providers. Gemalto can offer a consulting role in this definition process. These services are based on:

Standard GSM, and Telecom features and functions:

- Toolkit functions
- Functions specific to the operator or service provider
- Over The Air services

The card supports the APDU commands defined in 3GPP 51.011, as well as toolkit commands defined in 3GPP 51.014.

2.1.2 Service development

An operator, a service provider, a software house or Gemalto is allowed realizing the service development. Development is easily achieved by performing the following steps:

- Prototype the service.
- Program and compile an application in any standard Java™ development environment.
- On your PC, prepare the compiled application for loading onto the card. Define any application specific files (e.g., application PIN code file) and configure the application data.
- From your PC, load the application, its files and its configuration data onto the card.

2.1.3 Application validation

Before an application is deployed in the field to subscribers it must be certified to ensure that it is bug free and respects the functional and security requirements of the operator. Currently it is anticipated that only the operator or Gemalto can perform the role of certifying applications.

2.1.4 Application deployment

Applications can be deployed at any time during the life cycle of the card. Applications can be loaded:

- By Gemalto at the factory

- In the mobile using OTA customization
- From a terminal at the point of sale, from a PC via the web…

2.1.5 Application maintenance

Application maintenance can be broken into three categories:

- Modification of application data (e.g., menu texts, send SMS contents, text to be displayed)
- Modification of application files (e.g., application PIN code value)
- Update of application functionality (e.g., add a new menu). In this case the applet code has to be deleted and the updated code loaded.

In the same way as applications can be loaded, they can be deleted. The card allows application free space to be re-used thanks to the DMM mechanism (see the User's guide).

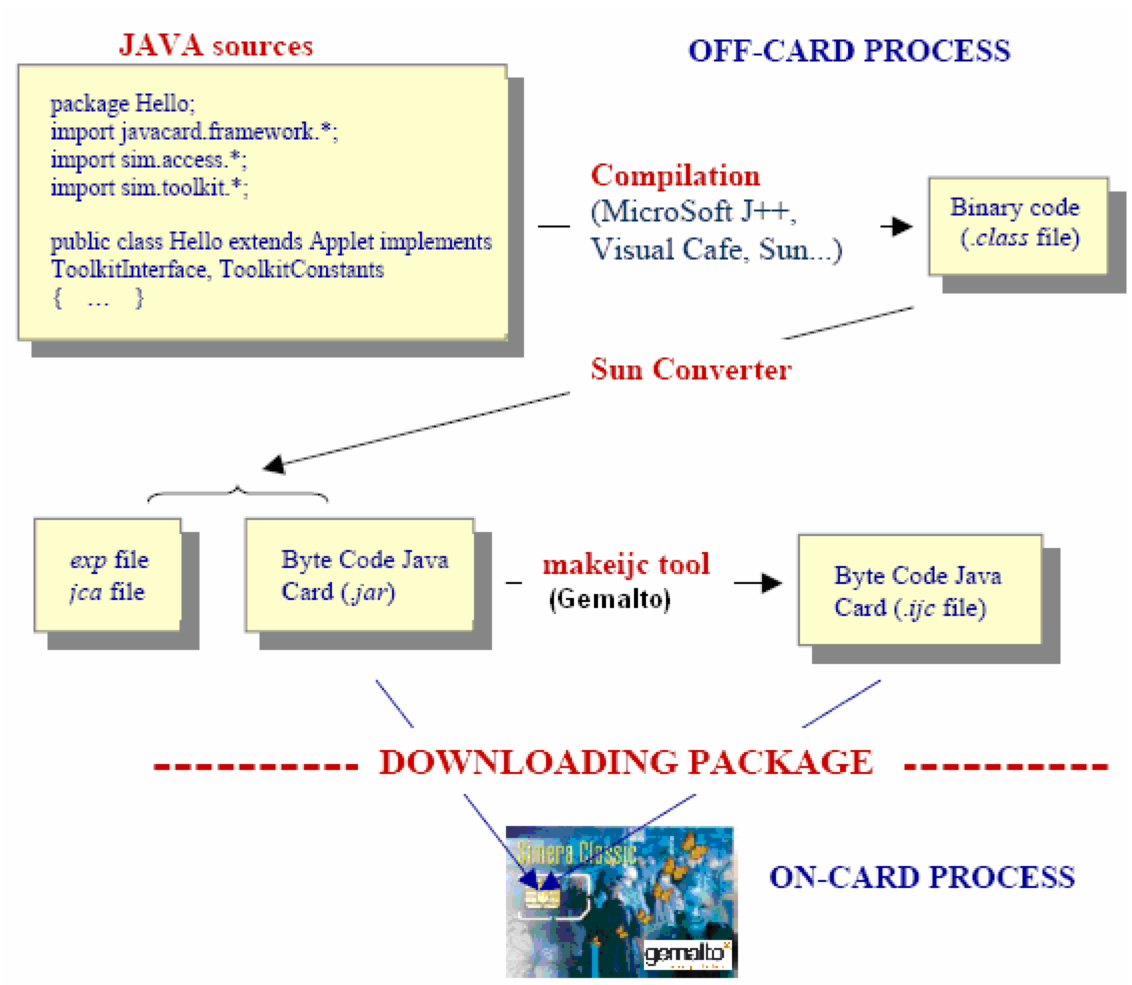
2.2 Applet Development Stages

An application passes different distinct stages of development, and life cycle states within the card. These are divided into two phases, off-card and on-card, related to the physical environment.

2.2.1 Off-card applet development

The off-card phase includes:

- Developing in Java Card™ language with the client terminal program (source code java file)
- Compiling (byte code class file and exp files)
- Converting into a format that can be loaded into Java™ cards (JAR or IJC files).



JAVA sources
OFF-CARD PROCESS
package Hello;
import javacard.framework.*;
import sim.access.*;
import sim.toolkit.*;
public class Hello extends Applet implements ToolkitInterface, ToolkitConstants
{ ... }
Compilation
(MicroSoft J++, Visual Cafe, Sun...)
Binary code (.class file)
Sun Converter
exp file
jca file
Byte Code Java Card (.jar)
makeijc tool
(Gemalto)
Byte Code Java Card (.ijc file)
DOWNLOADING PACKAGE
ON-CARD PROCESS

Figure 1 - Off-card applet development

2.2.2 Java Card™ source code

An applet loaded into the card provides the end-user with a set of services. These are activated in response to requests from the end-user or the client terminal. The client application in the terminal interacts with the applet as follow:

- The client application implements functions

- The application uses APDU commands to send parameters to the applet.
- The applet sends a response that is received by the client application.
- The client application analyses the card's response.

The source code can define two sorts of applets:

- The Java Card™ applets, created in compliance to the Java Card™ standards v 2.1.1. These applets use the Java Card™ APIs, which provide basic services common to any Java Card™ applet, such as:
  - Handling the APDUs independently from the protocol used (T = 0 or T = 1)
  - Managing security (signature, encryption, decryption, decryption, key, PIN)

  The APIs also provide utility services such as transactions or transient object creation.

  In the rest of this documentation, this kind of applet will be called "Java Card™ applet".
- The Java Card™ SIM Toolkit applet is a Java Card™ applet created in compliance to the TS 43.019 standard. This standard extends the Java Card™ v2.1.1 standards to allow the SIM card issuing pro-active commands and performing actions on the file's system.

  In the rest of this documentation, this kind of applet will be called "STK applet".

  The term "applet" will be dedicated to any of the kinds described above.

Within the source code, the following values have to be defined:

- Application Identifier (AID) is used for matching the applet with its client applications.
- CLA and INS bytes are used in APDU commands invoked by the client's terminal methods. These methods and APDU commands depend on the applet's functionality.

Application Identifier

Both packages and applets are identified with their application identifier (AID). This is included in the code of the applet and the client application, enabling a client application to target its corresponding application.

The AID is defined in ISO 7816-5. It is a byte string from 5 to 16-byte length. The identifiers are administered by ISO. You must register an application with ISO in order to get an official and unique AID. The application provider may also obtain a registered application provider identifier (RID) from ISO. The AID is constructed as shown in the following figure. The RID bytes are followed by 11 bytes that can be freely assigned by the application provider.

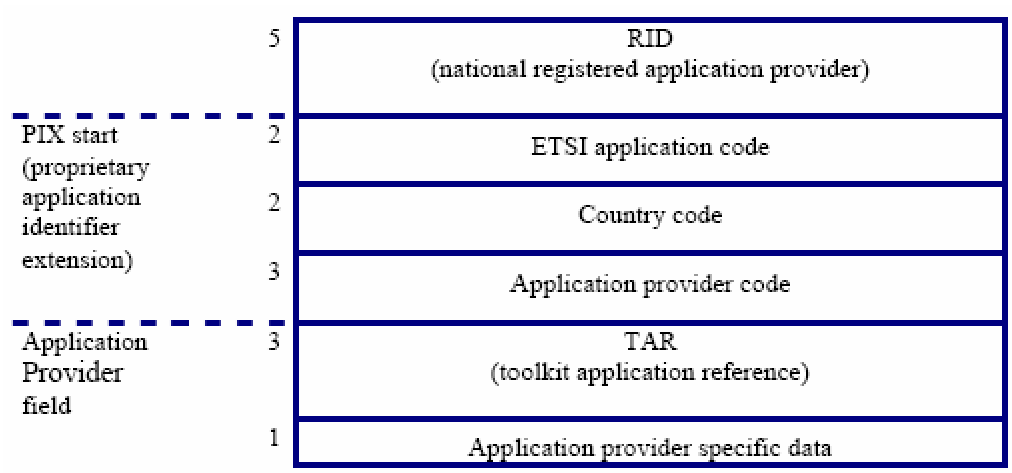

5
RID
(national registered application provider)
PIX start
(proprietary
application
identifier
extension)
2
ETSI application code
2
Country code
3
Application provider code
Application
Provider
field
3
TAR
(toolkit application reference)
1
Application provider specific data

Figure 2 - Structure of GSM AID

CLA and INS bytes

CLA byte represents a class of instructions. During execution, the applet verifies the CLA byte of an APDU command. Refer to ISO 7816-3 for more information about the class byte definition.

INS byte represents a particular instruction. It is declared at the beginning of the client application, which defines all the functionality of the applet.

Compiling source code

The source code must be compiled with a Java Card™ compliant compiler. The compiled source code is known as the binary code (class file).

Converting binary code

The binary code must be converted with a Java Card™ compliant converter. It transforms the binary format (class file) into a file format that can be loaded into the card. The converter allows to either convert a package alone (with no applet inside) or to convert a package and set the applet AID of the class that defines the applet.

Note: some tools (for example **jcconverter** and **jar2cap**) can be used to optimize the conversion.

The conversion is a two-step process that produces first a jca file (Java Card Assembly file) and an exp file (export file), then the jar and cap/ijc files (also known as Load File or package).

If you are using the **Gemalto Developer Suite**, (section 2.5) these conversion tools are embedded into the tool.

JCA, JAR, CAP and IJC files

The jca file is an ASCII file to aid testing and debugging. It is used to produce the jar file, created when a package of classes is converted.

The jar file is the Java Card byte code to be loaded in the card. It can be seen as a “zip” file, which contains a set of CAP files (Converted Applet files). Each CAP file is a component, which describes an aspect of the executable code to be downloaded.

IJC tools are widely available. If you are using the **Gemalto Developer Suite**, (section 2.5) there is an embedded component **jar2ijc** which is used to generate your IJC file.

Since IJC files are much smaller in size this allows gaining bandwidth when downloading **Over The Air**.

Note: the ijc file concatenates the CAP files in one file only, following the reference component order (described in the JCVM 2.1.1 standard) to produce a binary file directly loadable on the card.

EXP file

The exp file is created when the package is converted. It contains the information of the public interface for the package of classes. The exp file is not used directly by the Java Card™ Virtual Machine to execute the code. Nevertheless, it can be useful later to convert another package that imports classes from the first package.

Note: if the package does not contain toolkit applet, the export file contains all public or protected static methods and fields. If the package contains a toolkit applet, the exp file contains the shareable interface, javacard.framework.Shareable or an interface which implements the Shareable interface.

2.2.3 On-card process

The Card Manager is responsible for all application installations. It owns and uses a card registry, which holds information about the application life cycle states (see also the part “Setting applet status”). The on-card process includes:

- Loading the package
- Installing an applet instance from the package
- Making this instance selectable

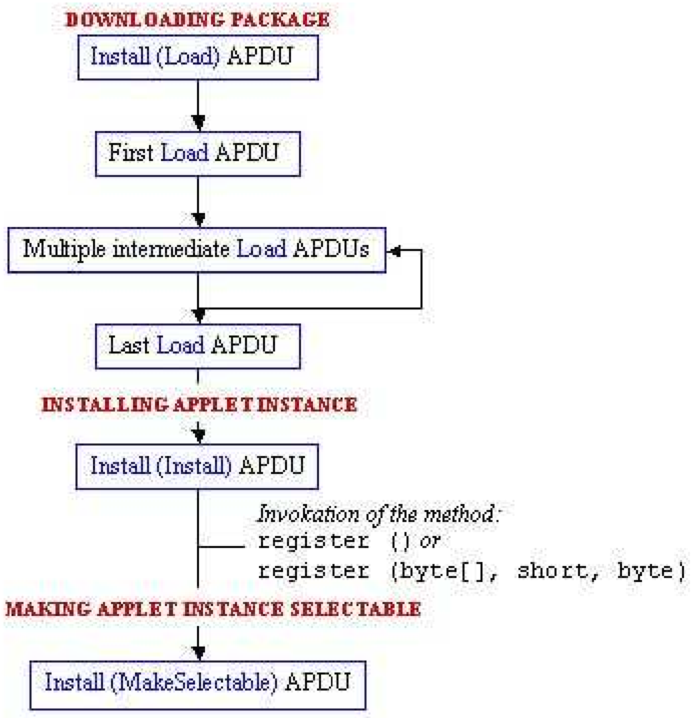

DOWNLOADING PACKAGE
Install (Load) APDU
First Load APDU
Multiple intermediate Load APDUs
Last Load APDU
INSTALLING APPLET INSTANCE
Install (Install) APDU
Invokation of the method:
register () or
register (byte[], short, byte)
MAKING APPLET INSTANCE SELECTABLE
Install (MakeSelectable) APDU

Figure 3 - On-card process

Note that the on-card process is aborted and any partially downloaded package or installed applet instance is deleted in the following cases:

- Access conditions are not fulfilled.
- Allocated EEPROM space is not sufficient.
- An exception, reset, or power fail occurs while a step of the process is not ended.

Loading the package

Loading the package consist in loading a Load File (or package). This file is composed of two blocks:

- One DAP (Data Authentication Pattern) block (optional) mandated for the Card Manager to perform a verification procedure (see below).
- One Load File Data Block, which contains the content of the ijc file.

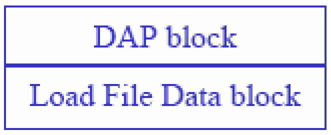


Figure 4 - Load File structure

The Load File is loaded by the use of two APDU commands:

- Install APDU command in *Load* mode

- Load APDU command

If the EEPROM memory space is insufficient, the DMM mechanism is automatically triggered to recover the memory needed.

Install (Load) command

The Install (Load) APDU instructs the Card Manager to initiate the loading and reserve the on-card resources. More precisely, the data field of this command indicates:

- The AID of the Load File. This value is checked by the Card Manager in order to ensure that no other on-card entity has the same AID value.
- The presence of the Load File DAP (optional property; do not confuse with the DAP block!) used to ensure integrity of the Load File content. If present, the Card Manager calculates a checksum, once the last block is received (last Load APDU command is received), by using a XOR algorithm on 16 bits. The result is compared to the value of the Load File DAP.
  - If the values are identical, the loading session continues.
  - If the values are different, the loading session is aborted and the Card Manager recovers the on-card resources.
- The size of the EEPROM memory (non-volatile memory) that the Card Manager must reserve to load the Load File; that is the size needed to load the ijc file, the static objects declared in the Load File (if present) and the DAP block (if present). If any, the DMM mechanism is automatically triggered to recover the EEPROM memory needed. Note:
  - If this size value is set to 0, the Card Manager automatically reserves the minimum memory needed once the ijc file is fully loaded (last Load APDU command is received).
  - Using the makeijc tool with the –verbose option allows retrieving the size of the static objects.
- Optionally, the size of EEPROM & RAM memories to request the Card Manager to check memories is sufficient to install and run an applet instance from the loaded package.

Load command

The Load APDU command is used to download the Load File; that is the DAP block (if any) and the Load File Data block (content of the ijc file).

Caution: this command must immediately follow the Install (Load) command, else the Load File loading is aborted, and the Card Manager recovers all the on-resources previously allocated.

According to the size of the Load File, several Load commands can be needed. The blocks can be downloaded in any order except the first one, which must be received in first (), and the last one, which must be received last. Once all blocks are received and stored, the Card Manager proceeds to the mandated DAP verification procedure (if a DAP block is present) and the package linking.

Note: the Card Manager is personalized such as the mandated DAP verification is mandatory or optional. In the first case, the Card Manager aborts any package loading if the DAP block is not included in the Load File, and an error Status Word is returned to the Load command. In the second case, the Card Manager allows the package loading, even if the DAP block is not include in the Load File.

The Card Manager proceeds to the DAP verification by using the DAP key coded on 8, 16 or 24 bytes and associated to a DES/TDES algorithm in Cipher Block Chaining (CBC) with ICV of 8 bytes of binary zero.

Note: the DAP key has the key index 10*h* and is stored in ONE key set whose version is defined during the card's personalization phase.

Installing an applet

Once the package is downloaded in the card, one or several applet instances can be installed. Fully installing an applet instance can be seen as a three-step process:

- Creating an applet instance by the way of the Install APDU command in Install mode.
- Registering the applet instance.
- Make the applet selectable by the way of the Install APDU command in Make Selectable mode.

Creating an applet instance

The Install (Install) command requests the Card Manager to create ONE applet instance at once from a downloaded package, and to allocate the on-card resources needed by this instance. The on-card resources are:

- EEPROM memory (non-volatile memory) to store persistent objects.
- RAM memory (volatile memory) used to store the content of transient objects.
- One free entry in the card's registry.

Note: the GetData APDU allows retrieving the maximum amount of EEPROM memory the Card Manager can allocate to the persistent objects.

The data field of the Install (install) command differs according to the kind of applet instance:

Parameters common to any applet instance:

- The AID value of the Load File, to allow the Card Manager identifying the package from which the applet instance must be installed.
- The AID within the Load File to allow the Card Manager retrieving the AID of the applet defined in the Load File.
- The AID of the applet instance to be installed (optional feature). The Card Manager checks this value is not already used, as two on-card entities cannot have the same AID value in the card's registry. If this occurs, the command is aborted and the Card Manager recovers all the on-card resources.
- The size of the EEPROM memory (non-volatile memory) to be reserved to store the persistent objects that the applet instance must create.
- RAM memory (volatile memory) to be reserved to store the content of the transient objects created by the applet instance. The Card Manager checks the free transient memory is sufficient before reserving it. If it is not the case, the command is rejected.

Parameters specific to STK applet

When a STK applet instance is installed, some parameters specify the ME and SIM resources that the applet can use:

- The Access Domain

  The Access Domain specifies the identities or access rights (CHV & ADM) granted to the STK applet instance to access to the GSM files and perform actions on these files according to their access conditions.

  - If all the access rights are granted, all actions are allowed except the ones with the NEVER access conditions
  - If no access right is granted, no action is allowed.
  - If some access rights are granted, only the actions performed by the defined

identities are allowed.

Defining an Access Domain for an applet instance is peculiar to this applet instance. This implies:

- Two STK applets instance installed from a same package can have different access rights on the GSM system
- The access rights of the STK applet instance are independent from the access rights granted to an entity (user, OTA message) at the SIM/ME interface (see the key set definition in the User's guide). For example, modifying the status of the CHV1 (disabled, blocked) defined for the user at the SIM/ME interface does not affect the CHV1 access right granted to a STK applet instance.

- The priority level

The priority level defines which STK applet instance must be activated when two or more STK applet instances have been registered to the same event. If STK applet instances are registered to the same event with the same priority level, then the last instance installed (the most recent one) is activated first.

- The number of timers (maximum of 8, else the card returns the Status Word 6A80 "incorrect parameters", see the TS 51.011 standard)
- The identifier of the menu entries

A unique item identifier can be defined for each STK applet instance to allow activating the STK applet from its ME menu entry.

The card automatically allocates the position of the menu entries. The order of the menu entries corresponds to the order of the registry entries which are free when the installation occurs. For example, if the STK instance A is installed before the STK instance B, its registry entry appears before the registry entry of the STK instance B. If an STK instance C is installed and the STK instance B is removed, then the registry entry of the STK instance A appears before the registry entry of the STK instance C and a free registry entry is located between those of A and C. If the STK instance D must be installed, the card searches for the first free registry entry that is, in this case, the entry previously occupied by the STK instance B. So the order of the menu entries becomes A D C.

- The minimum security level

The minimum security level defines which level of security must be applied on the OTA Command packets sent to the STK applet instance. This allows the STK applet instance, once installed and selectable, checking the security level of each of the OTA message it receives.

- If the security level corresponds to the minimum required, the security of the message is checked, and the STK applet instance processes the data.
- If the security level does not correspond to the minimum required, the STK applet instance rejects the OTA message. If required, a Response packet is returned to the server.

Registering an applet and making it selectable

Registering the applet instance is mandatory to successfully end its installation and to allow it being selectable.

The installation of the applet instance is considered complete upon successful return from one of the following methods:

- register () method (used when the AID of the instance defined via the Install (Install) APDU is the same as the AID within the Load File)
- register (byte [], bArray; short bOffset, byte bLength) method (used when the AID of the instance is provided in the INSTALL command parameters)

The applet instance is always registered with the value of the AID applet instance optionally defined via the Install (Install) APDU.

Using the Install (make selectable) APDU instructs the Card Manager to allow the selection of an applet instance via the SelectApplication APDU, and if it is a STK applet, from the triggering by event.

Allocation of the on-card resources

Each new instance reserves spaces in the EEPROM and possibly in the RAM. This depends on the content of the objects used by the applet instance when it is activated.

- The content of an object is persistent if it remains unchanged on the card from one session to another. This content is stored and referenced in the EEPROM memory as long as the applet instance is not deleted.
- The content of an object is transient when it does not persist from one session to the next, and are reset to a default state at specified intervals. This allows using it to store session keys. The content of a transient object is cleared to its default value (zero, false or null) at the occurrence of the following events:
  - CLEAR_ON_RESET – the object content is used for maintaining states that must be preserved across applet instance selections, but not across card resets.
  - CLEAR_ON_DESELECT – the object content is used for maintaining states that must be preserved while the applet instance is selected, but not across applet selections or card resets.

The transient contents are stored in RAM, but their references are in EEPROM.

The Card Manager allocates free transient memory differently according to the transient object contents:

- The CLEAR_ON_RESET object contents can never share RAM space. This means the Card Manager deems the RAM space is not available anymore once the space is already allocated to any of the transient object contents.
- The CLEAR_ON_DESELECT object contents can share RAM space if and only if the applet instances have not been installed from the same package.

When a new applet instance is installed, the Card Manager checks the free transient memory of the RAM is sufficient to store the transient object contents of this new instance.

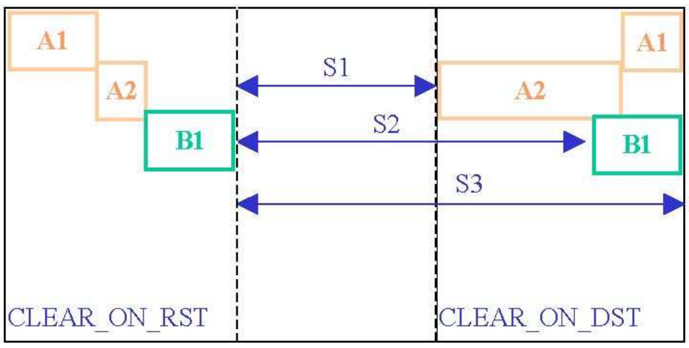


Figure 5 - RAM space reserved to the transient object contents

For example, from the figure above:

- If a new applet instance A3 must be installed, the S1 size is the free transient memory checked by the Card Manager.
- If a new instance B2 must be installed, the Card Manager checks the free transient memory needed is not superior to S2, knowing that the CLEAR_ON_RESET object content cannot use more than S1 free memory.
- If a new instance C must be installed, the Card Manager checks the free transient memory needed is not superior to S3, knowing that the CLEAR_ON_RESET object content cannot use more than S1 free memory.

2.2.4 Managing on-card applet instance

Setting applet status

After personalization phase, the Card Manager owns and maintains the card life cycle state information and manages the complete runtime environment. It is responsible for setting the following application life cycle states:

- Once the applet instance is installed on the card, its life cycle state is INSTALLED.
- To be executed the life cycle state of an applet instance must be SELECTABLE.
- To prevent any selection (and execution) of an applet instance, the life cycle state must be set to LOCKED.

The applet's life cycle state can be modified by use of the SetStatus command, with the P2 parameter indicating the new state:

- 03h = INSTALLED
- 07h = SELECTABLE. In this case, the applet instance is selectable via the SelectApplication APDU or can be triggered from an event. Note if the instance is registered to the MENU_SELECTION event, then it is included in the mobile menu presented to the subscriber.
- FFh = LOCKED. In this case, the applet instance is disabled. If it was registered to the MENU_SELECTION event, then it is not included anymore in the mobile menu presented to the subscriber.

Once the applet instance is available for selection or triggering from the outside world, it takes control of managing its own life cycle. Nevertheless the Card Manager can still take control of the application life cycle if a security problem is detected or if the applet instance has to be deleted.

Selecting a Java Card™ applet

A Java Card™ applet instance is selectable once it is correctly installed, registered within a unique Application IDentifier (AID) and in the state SELECTABLE.

The SelectApplication APDU command is used to select a Java Card™ applet internally to the card, with the data field of the command indicating the AID.

Once selected, the Java Card™ applet can access to the GSM file system and process the APDU commands.

Triggering a STK applet

A STK instance can be triggered once it is correctly installed, registered to events (see the next chapter) and in the state SELECTABLE.

When triggered, the STK instance can access to the file system and process actions on these files according to the access conditions defined on the files and the access domain defined for the STK instance. From another part, the STK instance can send pro-active commands according to the 3GPP 51.014 standard.

Note: when an applet is triggered via an OTA event, the initial context is restored including previously logged identity once data are processed.

Deleting applet instances or library packages

Removing on-card packages and applet instances is allowed with the DeleteApplication command. The command domain APPLET ADMIN must be granted to permit this action.

All the downloaded packages or installed applet instances can be deleted. The memory used by the package or the instance is recovered as a free space and can be reused for a new file/package/instance creation via the DMM mechanism (see the user's guide).

Note: a package cannot be deleted as long as one of its instances has not been deleted or if it is imported by another applet instance.

Retrieving applet data

The card can supply information. The APDU commands used to retrieve data are summarized in the following table:

APDU commands	**To retrieve data from:**
GetData	**Card Resources** The card stores information on card resources such as: ▪ Free EEPROM space ▪ Number of installed applets
GetData	**Menu parameters and menu texts (for STK applet instance only)** The card stores the positions and identifiers for the menu entries of the applets installed on the card: ▪ Item position ▪ Item identifier ▪ Item text Menu parameters are initialized during applet installation using the Install(install) APDU command. Menu parameters cannot be modified once the application has been installed. Menu entries are initialized by the applet using the sim.toolkit.ToolkitRegistry method initMenuEntry and can be changed by using the ChangeMenuEntry method.

GetData	**Application data of on-card resources (package or applet instance)** ▪ Type of the on-card resource ▪ Size of the on-card resource ▪ Length and value of the Card Manager AID ▪ Length and value of the package AID (applet instance only) ▪ Access Domain (STK applet instance only) ▪ Priority Level (STK applet instance only) ▪ Security Level (STK applet instance only)
GetData	**Package Class AID** List the AID of each applet class defined in a package.
GetStatus	**Life Cycle Status of the Card Manager and the on-card resources** • AID • Life cycle state • Privileges (for applet instances only)

2.3 Applet Development Environment

In this chapter, you will find some Java Card™ programming basics to allow you developing a simple application for the card. For a complete explanation, refer to the Java Card 2.1.1 and 3GPP 43.019 standards.

2.3.1 Why using JAVA™ development environment?

Java™ is an object oriented programming language developed by Sun Microsystems. This language is designed to be platform independent.

The Java™ language offers the following core advantages:

- A standard programming language – anyone who knows how to write a Java™ program can write a smart program and load it onto a card.
- Secure environment – Java™ is well known as a secure programming language.
- Multiple programs, or applets, on a card – the card architecture and the security features of Java™ language make it possible for multiple applets to reside safely on a card. The number of applets is only limited by the amount of space on the card.
- Full integration with mainstream Java™ IDEs – the card software integrates most Java™ integration development environments.
- All the benefits of object-oriented programming – programmers have the benefits of code reuse, design patterns, and superior structure.
- Platform independence – since Java™ smart card programs are portable across different chip architectures, applets cost less to develop and maintain.
- Dynamic updates – you can develop and deploy applets incrementally, adding features as you go along. You can add or delete the applets on a card at any point of its life cycle.

2.3.2 Java Card™ security

The integrity and security of Java™ are widely recognized. The security management developed for smart cards is implemented by the JCVM. The following features provide program and data integrity and security from malicious programs:

- The Java Card™ language is provided by the class file verifier, which is made off-card, before code is downloaded into the card.
- JCRE security enforces firewalls to isolate applets, which prevents unauthorized access of objects created by one applet from being used by another.
- Java Card™ compilers provide extensive stringent error checking when the program is compiled.

For example: all references to methods and variables are checked to make sure that the objects are of the same type. The compiler also ensures that a program does not access any non-initialized variables.

- All accesses to methods and instance variables in a Java Card™ class file are through access modifiers. These modifiers define a level of access control for each method. You can declare a method to be public (no limitations) protected (accessible by methods in the same subclass or package) or private (no access by other classes). If no declaration is made, the default allows the method to be accessed by any class in the same package.
- Basic Java Card™ types and operations are well defined. All primitive types have a specific size and all operations are performed in a designated order.
- Malicious programs cannot forge pointers to memory because there are no pointers that can be accessed by programmers or users.
- Additionally Java Card™ accesses variables only through references to them from the Java™ stack. Malicious programs are prevented from “snooping” around in the Java Card™ variable heap because the values of the local variables are unavailable after every method invocation. A method cannot access resources it shouldn’t.

2.3.3 Applet runtime environment

The Java Card™ technology defines a Java Card™ Runtime Environment (JCRE) that contains the full runtime environment to support the execution of Java Card™ program. The JCRE contains the Java Card Virtual Machine (JCVM) and provides classes and methods (API) to help developers create applets.

2.3.4 Java Card™ Virtual Machine

The Java Card™ Virtual Machine (JCVM) is a version of the Java™ Virtual Machine (JVM) adapted for smart cards. It controls access to all smart card resources, such as memory and I/O and allows applications to be securely loaded to the card post-issuance.

The JCVM executes the Java™ byte code subset on the smart card, ultimately providing the functions accessible from outside, such as signature, log-in and applications.

2.3.5 Java Card™ API

The available Application Programming Interface (API) classes allow developing applications and provide system services to those applications. These classes define the conventions by which a Java Card™ applet accesses the JCRE and native functions, including operating system functionality, memory access, and I/O operations. The APIs used by the card contains four packages:

- **javacard.framework** - this package contains the basics features needed to work with the Java Card™ card.
- **java.lang** - this package contains all the exceptions corresponding to a misusage of arrays, casts, and security. It is automatically imported by the compiler itself.
- **javacard.security** - this package contains a framework for the cryptography functions supported on the card.

- **javacardx.crypto** - this package contains a cipher class with encryption and decryption capabilities.

2.3.6 3GPP TS 43.019 API

These API classes are an extension of the Java Card 2.1.1 API classes. They allow application programmers accessing to the functions and data described in TS 51.011 and 3GPP 51.014, such as the SIM based services can be quickly developed and loaded onto SIM cards. The APIs used by the card contains two packages:

- **sim.access** - this package provides the means to the applets for accessing to the GSM data and file system of the GSM application defined in the TS 51.011 standard.
- **sim.toolkit** - this package provides the means for the toolkit applets to register to the events of the toolkit framework, to handle TLV information and to send proactive command according to the 3GPP TS 51.014 specification.

2.3.7 JCRE support services

The Java Card™ applets do not directly receive the incoming messages. These are first processed by the JCRE, which calls upon a method of the applet to process the APDU commands. The JCRE supports services dedicated to smart cards.

- It allows *applet isolation* thanks to applet firewall ensuring that no other applet may use, access or modify the contents of an object owned by another applet, except as defined by the applet itself.
- It includes a way to *share objects* between applet object sharing. An applet may permit restricted or unrestricted sharing of any of its objects. In another terms, any applet cannot access the fields of or objects of another applets implemented. There is an exception when the other applet explicitly provides interface for access.
- It includes a way to *manage atomic transactions*. A transaction is a logical set of updates of a persistent object. The transaction is atomic when all the fields are updated or none are. The mechanism of an atomic transaction allows protection against events such as power loss in the middle of the transaction, and against program errors that may cause data corruption. If the transaction cannot complete, the card data are restored to their pre-transaction states, at the exception of the content of the transient objects.

2.4 Writing a Toolkit Applet

The Java Card™ technology allows applets written in the Java™ language to be executed on a smart card. It defines a Java Card™ Runtime Environment (JCRE) and provides classes and methods to help developers create applets. In addition, the 3GPP TS 43.019 standard provides classes and methods to create toolkit applets; that are applets triggered at the following of a toolkit event selection.

2.4.1 Overview of a toolkit applet architecture

A toolkit applet class must extend from the javacard.framework.Applet class. This class is the super class for all the applets residing on a Java Card™. It defines the common methods an applet must support in order to interact with the JCRE during its lifetime. The class Applet provides a framework for applet execution. Methods defined in this class are called by the JCRE when it receives APDU commands.

To be considered as a toolkit applet, a Java Card™ applet must implement:

- The ToolkitInterface interface to be able to process toolkit events described in the TS 43.019 standard
- The ToolkitConstants interface to have access to constants specified in 3GPP 51.014 standard (for example: tags of the TLV/BerTLV, General results…).

After the applet code has been properly loaded and linked with other packages on the card, the applet's life starts when an applet instance is created and registered with the JCRE's registry table. A toolkit applet must implement the static method install() to create an applet instance. It must also register the instance with the JCRE by invoking one of the two register() methods and register the toolkit events by invoking the ToolkitRegistry() method.

On the card, a toolkit applet is in an inactive stage until it is explicitly triggered via an event.

Once the applet is triggered, the SIM toolkit framework calls the applet's processToolkit() method. This method processes the current toolkit event. If an error occurs, the throws() method of the ToolkitException interface specifies the reason of the exception. Finally, the applet returns in an inactive stage.

2.4.2 Example of applet

This applet has been designed to test the Call Control facility on a Mobile. It has one menu entry. When the user selects this entry, the applet will display a menu asking the user if he wants to activate the Call Control Facility (1) or De-activate it (2). If the user selects (1), the applet will register to the EVENT_CALL_CONTROL_BY_SIM else it will not register to the later by calling the clear method. If the facility is activated, when the user will set up a call, the applet will be triggered and will modify its call (result = 02) providing a new number (NUMBER which is a constant).

```
/*
 *  Copyright (c) Gemalto, unpublished
 *  work, created 2000. This computer program includes
 *  Confidential, Proprietary Information and is a Trade
 *  Secret of Gemalto Technology Corp. All use,
 *  disclosure, and/or reproduction is prohibited unless
 *  authorized in writing.
 *  All Rights Reserved.
 *
 */
package CallControl;

import javacard.framework.*;
import sim.access.*;
import sim.toolkit.*;
```

```
// You declare the class CallControl.
// This class inherits from the superclass Applet issued from
// the Java card library 2.1.1 and implements the methods
// defined in the interface ToolkitInterface and
// ToolkitConstants issued from TS 43.019.

public class CallControl extends Applet implements ToolkitInterface, ToolkitConstants {
  private ToolkitRegistry reg;
  private short menuId;
private static final short RES_CMD_PERF = (short)0;

  // The number to be call: 06 17 86 85 17
  private static final  byte [] NUMBER_BY_CONTROL={(byte)0x81,(byte)0x60,(byte)0x71,
(byte)0x68,(byte)0x58, (byte)0x71};

  private static final  byte [] NUMBER_CALLED={(byte)0x81,(byte)0x10,(byte)0x74,
(byte)0x64,(byte)0x35,(byte)0x07};

  private static final byte [] ALPHA_ID_1 = {(byte)'C',(byte)'a',(byte)'l',(byte)'l',
(byte)' ,(byte)'r',(byte)'e',(byte)'p',       (byte)'l',(byte)'a',(byte)'c',(byte)'e',
(byte)'d'}

  private static final byte [] ALPHA_ID_2 = {(byte)'A',(byte)'p',(byte)'p',(byte)'l',
(byte)'e',(byte)'t'};

  private static final  byte [] MENU_TITLE= {(byte)'C',(byte)'a',(byte)'l',(byte)'l',
(byte)' ',(byte)'C',(byte)'o',(byte)'n',       (byte)'t',(byte)'r',(byte)'o',(byte)'l'};

  private static final  byte [] EXITING = {(byte)'E',(byte)'x',(byte)'i',(byte)'t',
(byte)'i',(byte)'n', (byte)'g'};

  private static final  byte [] ACTIV = {(byte)'C',(byte)'a',(byte)'l',(byte)'l',(byte)' ',
(byte)'C',(byte)'o',(byte)'n',(byte)'t',       (byte)'r',(byte)'o',(byte)'l',(byte)'',
(byte)'a',(byte)'c',(byte)'t',(byte)'i',       (byte)'v',(byte)'a',(byte)'t',(byte)'e',
(byte)'d'};

  private static final byte [] ALREADY_ACTIV = {(byte)'C',(byte)'a',(byte)'l',(byte)'l',
(byte)' ',(byte)'C',(byte)'o',(byte)'n',        (byte)'t',(byte)'r',(byte)'o',(byte)'l',
(byte)' ',(byte)'a',(byte)'l',(byte)'r',        (byte)'e',(byte)'a',(byte)'d',(byte)'y',
(byte)' ',(byte)'a',(byte)'c',(byte)'t',        (byte)'i',(byte)'v',(byte)'a',(byte)'t',
(byte)'e',(byte)'d'};

  private static final byte [] DESACTIV = {(byte)'C',(byte)'a',(byte)'l',(byte)'l',
(byte)' ',(byte)'C',(byte)'o',(byte)'n',       (byte)'t',(byte)'r',byte)'o',(byte)'l',
(byte)' ',(byte)'D',(byte)'e',(byte)'-',       (byte)'a',(byte)'c',(byte)'t',(byte)'i',
(byte)'v',(byte)'a',(byte)'t',(byte)'e',       (byte)'d'};

  private static final byte [] ERROR = {(byte)'E',(byte)'r',(byte)'r',(byte)'o',(byte)'r'};

  private static final byte [] SELECT_ITEM = {(byte)'C',(byte)'a',(byte)'l',(byte)'l',
(byte)' ',(byte)'C',(byte)'o',(byte)'n',       (byte)'t',(byte)'r',(byte)'o',(byte)'l'};

  private static final byte [] ACTIVATE = {(byte)'A',(byte)'c',(byte)'t',(byte)'i',
(byte)'v',(byte)'a',(byte)'t',(byte)'e'};

  private static final byte [] DEACTIVATE = {(byte)'D',(byte)'e',(byte)'s',(byte)'a',
(byte)'c',(byte)'t',(byte)'i',(byte)'v',       (byte)'a',(byte)'t',(byte)'e'};

  private static final byte [] SET_UP_CALL = {(byte)'S',(byte)'e',(byte)'t',(byte)'u',
(byte)'p',(byte)' ',(byte)'C',(byte)'a',      (byte)'l',(byte)'l'};

  private static final byte [] PROFILE =
   {
      (byte)0,     //  PROFILE DOWNLOAD
      (byte)3,     //  MENU SELECTION
      (byte)8,     //  COMMAND RESULT
      (byte)9,     //  CALL CONTROL BY SIM
      (byte)12,    //  HANDLING OF THE ALPHA ID
      (byte)16,    //  DISPLAY TEXT
      (byte)24,     //  SELECTED ITEM
      (byte)28     //  SET UP CALL
   };

    // Constructor of your applet: create an object of your class
    // The constructor role is double. First it must register
    // the object, and second, it must define the menu
    // selection. Here, only one menu item is created.
  private CallControl(byte [] buffer, short offset, short length)    {
      // Register to the SIM Toolkit Framework
      reg = ToolkitRegistry.getEntry();
      // Define the menu entry
      menuId = (short)((short)0x00FF & reg.initMenuEntry(MENU_TITLE,      (short)0,
(short)MENU_TITLE.length, (byte)0,     false, (byte)0, (short)0));
      register();
   }

  // install method
  // This method allows creating the instance file. It
  // calls the constructor via the keyword: new.
  public static void install(byte [] buffer, short offset, byte length)
   {
      new CallControl(buffer, offset, (short)(length & 0x00FF));
   }

  // process method
  // This method (applet class) allows incoming APDU
  // commands for a Java Card applet only.
  public void process(APDU apdu)
   {
   }

// AT THIS STADE, THE APPLET IS COMPLETELY INSTALLED
   // processToolkit method (from ToolkitInterface, GSM
   // 03.19)
   // A toolkit applet uses this method to process the
   // current toolkit event.

  public void processToolkit(byte event)
   {
      EnvelopeHandler eh;
      EnvelopeResponseHandler erh;
      ProactiveHandler ph;
      ProactiveResponseHandler prh;
      short res;
      short i;
      byte [] msg;
      switch(event)
       {
       case EVENT_PROFILE_DOWNLOAD:
       for(i=(short)0; i < (short)PROFILE.length ;       i++}
       {
              if(!MEProfile.check(PROFILE[i]))
              {
                     reg.disableMenuEntry((byte)menuId);
                     return;
              }
       }
       reg.enableMenuEntry((byte)menuId);
       return;

    case EVENT_CALL_CONTROL_BY_SIM:
       erh = EnvelopeResponseHandler.getTheHandler();
       erh.appendTLV(TAG_ALPHA_IDENTIFIER, ALPHA_ID_1, (short)0,
(short)ALPHA_ID_1.length);
       erh.appendTLV(TAG_ADDRESS, NUMBER_BY_CONTROL, (short)0,
(short)NUMBER_BY_CONTROL.length);
           // Allowed with modification
       erh.postAsBERTLV(SW1_RP_ACK, (byte)0x02);
       return;

    case EVENT_MENU_SELECTION:
       ph = ProactiveHandler.getTheHandler();
       ph.init(PRO_CMD_SELECT_ITEM, (byte)0, DEV_ID_ME);
       ph.appendTLV((byte)(TAG_ALPHA_IDENTIFIER|TAG_SET_CR),      SELECT_ITEM, (short)0,
(short)SELECT_ITEM.length);
       ph.appendTLV(TAG_ITEM, (byte)1,  ACTIVATE,(short)0,        (short)ACTIVATE.length);
       ph.appendTLV(TAG_ITEM, (byte)2,  DEACTIVATE,(short)0,       (short)DEACTIVATE.length);
       ph.appendTLV(TAG_ITEM, (byte)3,  SET_UP_CALL,(short)0,
(short)SET_UP_CALL.length);
       res = (short)((short)0x00FF & ph.send());
```

```
        switch(res)
        {
                case RES_CMD_PERF:
                        prh = ProactiveResponseHandler.getTheHandler();
                        switch(prh.getItemIdentifier())
                        {
                                case (short)1:
                                        if(!reg.isEventSet(EVENT_CALL_CONTROL_BY_SIM))
                                        {
                                                reg.setEvent(EVENT_CALL_CONTROL_BY_SIM);
                                                msg = ACTIV;
                                        }
                                        else
                                        {
                                                msg = ALREADY_ACTIV;
                                        }
                                        break;

                                case (short)2: msg = DESACTIV;
                                        reg.clearEvent(EVENT_CALL_CONTROL_BY_SIM);
                                        break;

                                case (short)3: //  Set Up Call
                                        ph.init(PRO_CMD_SET_UP_CALL, (byte)0x00,
DEV_ID_NETWORK);
                                        ph.appendTLV(TAG_ALPHA_IDENTIFIER,        ALPHA_ID_2,
(short)0,         (short)ALPHA_ID_2.length);
                                        ph.appendTLV(TAG_ADDRESS, NUMBER_CALLED,       (short)0,
(short)NUMBER_CALLED.length);
                                        ph.send();
                                        return;

                                default:
                                        msg = ERROR;
                                        break;
                        }
                        break;

                default:
                        msg = EXITING;
                        break;
        }
        ph.initDisplayText((byte)0x80, DCS_8_BIT_DATA, msg, (short)0, (short)msg.length);
        ph.send();
        return;
 }
 }
}
```

2.4.3 Limitations in an applet creation

This part deals with some guidelines to increase the card resource memory. The following list is not exhaustive. Keep in mind that you must always test several possibilities to optimize the applet.

- Do not use:
    - Unicode character support
    - 32-bit and 64-bit integers
    - Float and double data types
    - Threads
    - Multidimensional arrays
- Create one class only.
- Create all objects in the applet's constructor.
- Clean up the code; that is, remove the methods, variables, operations that are not necessary in the applet.
- Factoring common code to eliminate redundancy in a method.
- Use primitive types. Avoid creating objects from primitive types. Developing a class to encapsulate primitive types can give more functionality than the primitive type itself, but it

can use memory resource of the card.

- Use constants, via the keywords static and final. Using static final improves both application size and performance.
- Avoid using local object or arrays. Each instance of an object or array declared with a local scope allocates memory. As a Java Card does not have a garbage collection, this memory is never freed. Each call to a local method allocates memory again, and soon or later uses up the memory resources.
- Re-use variable when possible because the more variables are used the more card resources are consumed.
- Use a variable to store an array element. Usually accessing array elements requires more byte codes than accessing local variable. If an array element is accessed multiple times from different locations in the same methods, save the array value in a local variable on the first access, then use the variable in the subsequent accesses.
- Gain on-card execution time by use transient objects to store intermediate results or frequently updated temporary data. Writing to RAM is 1,000 times faster than writing in EEPROM.
- Use the switch statement in place of the if-else statement. It often executes faster and takes less memory than the equivalent if-else.
- Use compound arithmetic statements instead of separate assignments. The reason is separate assignments require additional instructions to first store all intermediate values, then load them back for the next calculation.

2.5 Developer Suite as a Java Card IDE

The development of Java Card Applets and STK Applets is greatly facilitated by the use of the Developer Suite, a fully integrated Java Card IDE.

The Developer Suite – based on Eclipse Java IDE – will help you learn how to develop Java Card Solutions with Rapid Application Development Wizards for basic Java Card and STK Applets. The Suite also provides an End-to-end Simulation Environment, enabling complete testing of your Solution before you deploy it in the Real World.

A free evaluation of the Developer Suite is available on the Gemalto Developer Network: http://developer.gemalto.com.

3 Basic Development Rules

Failure to conform to the set of recommendations in this chapter may lead to basic development issues and lack of standard optimization.

D1 – Use APIs, rather than rewriting methods, whenever possible

Use APIs

Description :
Use APIs whenever it is available to avoid the need to duplicate the code. Moreover, the strongest mechanism provided by the API shall always be chosen. This holds for Java Card standard API, GP API and UICC API.

D2 – Coding and storing PINs and Keys in primitive arrays must be avoided

Coding PINs & Keys

Description :
To ensure security of the card is not compromised, coding / storing of PINs and KEYs must not be in primitive arrays. Below are examples of GOOD recommended practice.

Example :

```
OwnerPIN pin;
Pin = new OwnerPIN (tryLimitApplet, maxPINSizeApplet);
Pin.update( pinApplet offset, length )
```

```
Key rsaPrivateKey;
rsaPrivateKey = (RSAPPrivateKey) KeyBuilder.buildKey
  (KeyBuilder.TYPE_RSA_PRIVATE, KeyBuilder.LENGTH_RSA_512, False);
rsaPrivateKey.setExponent(bufferExp, offsetExp, lengthExp);
rsaPrivateKey.setModulus(bufferMod, offsetMod, lengthMod);
```

D3 – Sensitive data must be initialized at the beginning and clears at the end of the session

Sensitive data – initializations and clearing

Description :
Rule is applicable to global arrays, keys updates and Session Objects.

Always :

- Initialize at the beginning of the session
- **And clear** at the end of the session

Note :
To reset a key, always use the clearKey() method of the javacard.security.key interface. Do not explicitly erase the key with zeros or other value.
Rationale:

- Benefits from platform counter-measures.

- When using clearKey() method, the initialized state of the key is set to false.

Example :

```
sessionflags =
  JCSystem.makeTransientByteArray
    ((short)13,JCSystem.CLEAR_ON_RESET);
sessionflags =
  JCSystem.makeTransientByteArray
   ((short)13,JCSystem.CLEAR_ON_DESELECT);
```

```
sessionKey = (DESKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_DES_TRANSIENT_RESET,
  KeyBuilder.LENGTH_DES3_2KEY,
  false);
sessionKey = (DESKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_DES_TRANSIENT_DESELECT,
  KeyBuilder.LENGTH_DES3_2KEY,
  false);
```

The code in red will only reset the data on RESET. As a result, data is not cleared at the end of the select session.

The code in green shows you the correct setting

D4 – Sensitive data must be stored in transient data

Sensitive data – Storage

Description :
Store your session data in transient data. Avoid using the APDU buffer to store this data.

Example :

```
sessionflags =
  JCSystem.makeTransientByteArray
   ((short)13,JCSystem.CLEAR_ON_DESELECT);
```

Store your sensitive data in transient data.

D5 –Protect your sensitive data against Rollback attack

Counter management to combat against Rollback attack

Description :
An attacker can use the fact that your code is under transaction, and power off the card in order to roll-back to the old value. Counter management must be implemented to counter Rollback attacks when working with sensitive data.

Rule:
If a sensitive mechanism that relies on a ratification counter needs to be managed, always perform the following actions:-

- Check the ratification counter value and return error if set to 0.
- Atomically decrement the ratification counter
- Verify that the code is not under transaction
- Run the “try” function
- If the “try” is successful, atomically increment the ratification counter.

D6 - All initialized buffers (containing menu item, strings…) must be declared as static

Initialized buffers declared as static

Description :
All initialized buffers (containing menu item, strings…) must be declared as static:
- to save execution time during installation
- to save code space

to prevent errors during installation (transaction buffer full)

Example :

`public class exampleD1 extends Applet implements ToolkitInterface{`
 	
`byte[] menu1 = {(byte)'m', (byte)'e', (byte)'n', (byte)'u', (byte)'1' };`
 	
`static byte[] menu2 = {(byte)'m', (byte)'e', (byte)'n', (byte)'u', (byte)'2' };`
	For all initialized arrays, avoid the declaration in red, even if the syntax correct. Use the syntax in green instead.

D7 - Reentrance

Description:

A **proactive session**, initiated by an *APPLICATION A*, execution **is interrupted** when a second *APPLICATION B* (can be the same one) **is activated**.

After *APPLICATION B* **has been finished**, and no additional event occurs before the terminal response is received, **control is returned** to *APPLICATION A*, so that **its own execution can be finished**.

A **terminal CAN only manage one proactive command at a given time**, it **is not possible** for the *Application B* to **initiate a proactive session**.

If this application uses the ETSI 102 241 UICC API (Release 6) or the 3GPP 43.019 API (Release 5), a ToolkitException with the reason code HANDLER_NOT_AVAILABLE will be thrown when trying to retrieve the ProactiveHandler system instance.

How to deal with re-entrance?

- **Requiring an SMS-Submit**

  Can be used only: if the application has to send proactive commands only when triggered by EVENT_FORMATTED_SMS_PP_ENV.
  The remote server must require a response packet using SMS-Submit when sending a message to the application (the card shall implement the 3GPP 43.019 Rel-5)

  Advantages:
  - o it is a standard mechanism
  - o there is no need to implement anything related to this mechanism in the application

- **Throwing an ISOException**

This **Release-6 solution is quite simple**: the applet directly throws an ISOException with reason code ’93 00’ if it is not able to process the current message.
It may be very useful if there is no issues with formatted SMS and counter check, and could be used for instance on ENVELOPE (Timer Expiration), as the message will be sent at a later stage by the terminal.

- **Event Proactive Handler Available**

Simplest solution for the applet developer, as it does not have to check ProactiveHandler availability when triggered by this event.

If an applet **is not able to process a first incoming message** linked to the unavailability of ProactiveHandler, it has to do as follows:

1. Save needed data
2. Register the applet on EVENT_PROACTIVE_HANDLER_AVAILABLE
3. Exit.

When the applet is triggered on this event, it is sure that ProactiveHandler is available, and saved data can be processed.

This method is applicable on Release-6 cards

D8 – Handler availability

Description :

A minimum requirement for the **availability of the system handlers** and the **lifetime of their contents** are defined. *Please refer to 3GPP 31130 ((U)SIM API), or TS 101476 (GSM API).*

Example :

<table>
<tr>
<td>

```
public class exampleD6 extends Applet
implements ToolkitInterface{

  byte[] apduBuffer;

  void processToolkit(byte event){
    EnvelopeHandler hdlr;

    switch (event) {
      case PROFILE_DOWNLOAD:
          hdlr = EnvelopeHandler.getTheHandler();
      default:
          break;
  }
```

</td>
<td>The assignment in red is forbidden since the EnvelopeHandler is not available by the system during the profile download procedure.</td>
</tr>
</table>

D9 – Global and local variables

Global vs local variables

Description:

There are several types of variables in Java Card applet:

- **Instance variables (global variables):** created at the object instantiation.
- **Local variables**: they are accessible only from the function or block in which they are declared.

<u>Some basic rules:</u>

- **Not use instance variable when a static variable can be used**. It permits avoiding redundancies in class instances.

- **Minimize the number of instance variables**.

- **Declare your constants as static final**. (code size and performance optimization)

    - Static variables which are pre-defined in code and are not expected to change during the application life-time should be changed to "static final"
    - Arrays containing pre-defined values (like menu items, text messages, strings...) must be declared as static to optimize both the applet installation processing time and the EEPROM occupation

- **Do not declare too much arguments in a method** ➔ very costly in memory (RAM),

- **Local variables access are faster** than global variable access

    - Working variables (like loop counters, temporary storage variables...) must be declared as local variables of a method instead of global variables of a class, to avoid over-stressing the EEPROM
    - Working buffers must be declared as transient arrays (located in RAM) whenever possible, to avoid over-stressing the EEPROM. If not possible (lack of RAM resources), care must be taken on the working buffers update frequency (when and how many times)
    - RAM space should be used only when needed, as RAM is a costly resource. 5 bytes per kilo byte of used EEPROM seems a relevant ratio for cost efficiency.

- **Factorize arguments of a method** ➔ Use instance variables instead of parameters and Pass an object instead of a list of parameters

- **Maximize the local variables used** ➔ however reuse them as much as possible

The consequence of these recommendations is the reduction of the execution time.

<u>General advantage:</u>
Reducing the number and size of classes and objects facilitates the resolution of a bug ➔ easier readability.

Example :

```
~~public class MyApplet{~~
~~     private byte OFFSET_A = (byte) 1;~~
~~     private byte OFFSET_B = (byte) 2;~~
~~…~~
```

Static declaration: variable value may change over time

`}`
 	
`public class MyApplet{`
`    private static byte OFFSET_A = (byte) 1;`
`    private static byte OFFSET_B = (byte) 2;`
`    …`
`}`	

~~`public class MyApplet{`~~
~~`    private byte OFFSET_A = (byte) 1;`~~
~~`    private byte OFFSET_B = (byte) 2;`~~
~~`…`~~
~~`}`~~
 	
`public class MyApplet{`
`    private static final byte OFFSET_A =`
`    (byte) 1;`
`    private static final byte OFFSET_B =`
`    (byte) 2;`
`    …`
`}`
	Constant declaration: variable value will never change over time

`void init_my_buffer(byte[] buffer, short`
`length){`
`    short offset = (short)(length/2);`
`    Util.arrayCopyNonAtomic(buffer, 0, data_1,`
`    offset,(short)8);`
 	
`    offset = (short)( length/2+8);`
`    Util.arrayCopyNonAtomic(buffer, 0, data_2,`
`    offset,(short)8);`
 	
`    offset =(short)(length/2+16);`
 	
`    Util.arrayCopyNonAtomic(buffer, 0, data_3,`
`    offset,(short)16);`
`}`
	Reuse local variable

D10 – Storage of references to temporary entry point objects is forbidden

Temporary entry point objects reference storage

Description :
The Java Card™ standard specifies JCRE entry point objects (EPO). Among those EPOs the temporary EPO are prevented to be stored in class variables, instance variables or array components. The JCRE detects and restricts attempts to store references to these objects as part of the applet firewall functionality. A Security Exception is thrown by the system in that case.
Examples of main Temporary EPO

- APDU
- EnvelopeHandler, ProactiveHandler, EnvelopeResponseHandler, ProactiveResponseHandler.

Example :

```
public class exampleD8 extends Applet implements
ToolkitInterface {

  byte[] apduBuffer;
  byte[] envHandler;

  void process(APDU apdu){
    byte[] buffer;
    apduBuffer = apdu.getBuffer();
    buffer = apdu.getBuffer();
  }
```
	The assignment in red is forbidden by the system since it is an attempt to store the APDU buffer reference to an instance variable.
	
	The assignment in green is correct, since the storage is on a local reference, so released at the end of the process() method execution.

```
void processToolkit(byte event){
    byte[] handler;
    envHandler = EnvelopeHandler.getTheHandler();
    handler = EnvelopeHandler.getTheHandler();
}
```
	Same for the handlers: the assignment in red is forbidden by the system since it is an attempt to store the Envelope handler reference to an instance variable.
	
	The assignment in green is correct, since the storage is on a local reference, so released at the end of the processToolkit() method execution.

D11 – Keep methods sizes small

Keep methods sizes small

Description:
Size of methods should be small. A maximum of about 300 bytes for one method is acceptable. It is recommended to break-up into more methods.

General advantage:
Having smaller methods improves readability and ease maintenance and debugging.

D12 – Ensure number of methods do not exceed 256 methods per class

Ensure methods do not exceed 256 methods per class

Description:
When breaking up the methods, take care not to produce too many methods, because there is a limitation of 256 methods per class and secondly more methods will mean a bigger constant pool component, which will be completely moved to EEPROM when patching.

4 Recommendations to minimize risks in field

Due to the limited lifespan of the EEPROM and flash memory, applets that perform excessive read/writes repeatedly on the same location has the risk of stress failure in the field. This section covers the recommendations for in the applet design to minimize the need for read/write for a memory location.

Failure to abide to the recommendations in this chapter may lead to a severe field issue: the card can become mute or with unexpected behavior.

F1 - Check where all your variables are stored and accordingly move them if there is a risk of NVM stress (guaranteed 100,000 updates for E^{2}PROM and FLASH technologies)

Stress EEPROM : objects in EEPROM risk

***Description* :**
The Java and Java Card memory models are different: on a Java Card, objects are allocated in EEPROM. It means:

- all global variables of basic type (byte, short…)
- objects (created using the Java new instruction)

will be located in EEPROM.
The only memory chunks allocated in RAM are:

- the local variables,
- the parameters of methods

the transient objects explicitly created by using dedicated Javacard.framework.APIs

***Example* :**

```
public class exampleF1 extends Applet {

  byte counter=0;
  byte[] workingBuffer;
  byte[] transientBuffer;
  short i, j, k;
  private ToolkitRegistry reg;
```
	counter, workingBuffer, transientBuffer, i, j, k are global variables of basic types → located in EEPROM
	
	reg is an object of type ToolkitRegistry → located in EEPROM

```
void process(APDU apdu){
    short i, j, k;
    byte[] reference;
    …
  }
```
	apdu is a parameter of process method → located in RAM
	
	i, j, k, reference are local variables of basic type → located in RAM

```
public exampleF1 () {
    transientBuffer =
    JCSystem.makeTransientByteArray((short)10
    0, JCSystem.CLEAR_ON_RESET);
    workingBuffer = new byte[12];
  }
  …
}
```
	workingBuffer is a global variable located in EEPROM, referencing an object located also in EEPROM created with new.
	
	transientBuffer is a global variable located in EEPROM, referencing a transient array located in RAM, created using dedicated Java Card™ APIs.

F2 - Avoid writing in EEPROM upon STATUS event reception

Event status

Description :
An applet registered on the STATUS event will be triggered each time the handset issues a STATUS APDU command, i.e. every 1 or 2 minutes. Special care must be taken in the applet on the EEPROM parts (object fields, files…) that are written during that procedure.

Example :

```
public class exampleF2 extends Applet
implements ToolkitInterface{

  byte[] workingBuffer;
  byte[] transientBuffer;
  short s;

  public exampleF2() {
    transientBuffer =
    JCSystem.makeTransientByteArray((short)20
    0, JCSystem.CLEAR_ON_RESET);
    workingBuffer = new byte[200];
  }
```

```
  void processToolkit(byte event){
    short i;
    switch (event) {
      case EVENT_STATUS_COMMAND:
         for (s=0; s<200; s++) {
             workingBuffer[s]=(byte)s;
         }

      default:
         break;
    }
```

This routine in red is called every time a STATUS command is sent by the handset, ie every 1 or 2 minutes)
The routine fills up the buffer referenced by workingBuffer located in EEPROM.
Furthermore the index s used in the for loop is also located in EEPROM.
→ This routine potentially writes 200 in EEPROM on two EEPROM cells every 2 minutes, ie 144000 times a day
→ The EEPROM will be then killed in a few days.

```
  void processToolkit(byte event){
    short i;
    switch (event) {
      case EVENT_STATUS_COMMAND:
         for (i=0; i<200; i++) {
             transientBuffer[i]=(byte)i;
         }

      default:
         break;
    }
```

This routine in green is called every time a STATUS command is sent by the handset, ie every 1 or 2 minutes)
The routine fills up the buffer referenced by transientBuffer, which is located in RAM.
Furthermore the index i used in the for loop is a local variable so located in RAM.
→ This routine gives exactly the same result as the routine in red but does not write in EEPROM in any case.

Note : standard 102.241 release 6 defines a standard buffer in RAM accessible for applets (available in uicc.system.UICCPlatform, retrievable using **getTheVolatileByteArray**()). This buffer can be used as a temporary buffer in case the applet developer needs a temporary RAM space.

F3 - Avoid writing in EEPROM upon file update events when the concerned file is a HIGH update activity file

File Update

Description :
Same remark as for rule #2 above for the event FILE_UPDATE, some files (EFLoci, EFBCCH for instance) are defined in the 3GPP51.011 & 3GPP31.102 standard with a HIGH update activity. If an applet is triggered on such file update, care must be taken in the applet on the other EEPROM parts (object fields, files) that are written during that procedure.

Example :

Same example as in F2
Consider EVENT_FILE_UPDATE instead of EVENT_STATUS_COMMAND

F4.1 - Avoid passing too many parameters in a function (4 or 5 max)
F4.2 - Avoid declaring too many local variables inside each method
F4.3 - Reduce the overall nesting level (ie a function calling a function). Latest Gemalto Java cards allow more than 20 nesting levels

Warning functions calls / Jstack

Description :
A Java Card stores all methods parameters and local variables in a RAM dedicated area called the J-Stack.
On a Smartcard, the RAM resources are limited.

Example :

F4.1 Passing too many parameters	

```
void myMethod(   byte [] buf1
                 short off1
                 short len1
                 byte [] buf2
                 short off2
                 short len2
                 byte [] buf3
                 short off3
                 short len3
                )
```
	9 parameters declared + this (current object reference).
	
	→ try to reduce the number of parameters passed in a method

F4.2 Declaring too many local variables	

```
void myMethod (  byte [] buf1
                 short off1
                 short len1)
                )
short s1, s2, s3;
byte b1, b2, b3;
byte[] ba1, ba2;
Object o1, o2;
```
	10 local variables declared
	
	→ try to optimize the local variable usage in methods (reuse variable for several purpose for example)

F4.3 Nesting level	

`void myMethod1(byte [] buf1`
`                  short off1`
`                  short len1)`
`                ) {`
`     myMethod2();`
`}`
 	
`void myMethod2() {`
`     myMethod3();`
`}`
 	
`void myMethod3() {`
`     myMethod4();`
`}`
 	
`void myMethod4() {`
`     doSomething();`
`}`
 	
`void doSomething() {`
`     …`
`}`
	When calling myMethod1, the nesting level is 5 since mymethod1 invokes myMethod2() which invokes myMethod3() which invokes myMethod4() which invokes doSomething().
	
	Each method invocation creates information in the J- Stack, the J-Stack size in RAM being limited, so special care must be taken on the nesting level of the overall application.

F5 - ALL objects making up the application are created during installation

Objects creation

Description :
As opposed to Java, a Java Card does not necessarily implement a Garbage collector running in background for freeing the memory space taken by no longer referenced objects. Thus it is recommended to create all new objects at applet installation time (i.e. in the install() method or constructor).

Example :

`public class exampleF5 extends Applet {`
 	
`  byte[] workingBuffer;`	

`void process(APDU apdu){`
`   workingBuffer = new byte[12];`
`   …`
`  }`
`}`
	Object assigned to workingBuffer is created in process(): **an object will then be created AT EACH APDU SENT TO THE APPLET, so filling up the remaining free NVM.**

`public exampleF1 () {`
`   workingBuffer = new byte[12];`
`  …`
`  }`
	Object assigned to workingBuffer is created in the applet's constructor: **the object will then be created only once at applet installation time**
	
	This way is recommended
	
	Use the constructor or the applet's install() method.

F6 - If your application uses the Java Card transactions:

- **The transaction critical section (ie part between a Begin and a Commit) shall be as small as possible.**
- **The begin and commit shall be done in the same method.**
- **All execution paths of a method starting a transaction shall complete it.**
- **The transaction critical section shall be protected by a try-catch block**

Transactions (ex : Begin and Abort management with exception, …)

Description :
This section aims at preventing card tear out or power supply loss during an application execution.

See examples in the related *“development guidelines for anti-tearing”*

F7 - Insert a MORE TIME proactive command if your toolkit process is longer than 2s in order not to block the handset

Verification that the execution time is < 2s : More Time / Trig applet
Description :
The SCP102.221 standard specifies that a long Card Application Toolkit process may prevent the handset from sending “normal GSM commands” which are time critical (authentication…), and therefore advises the application to send MORE TIME proactive commands not to block the handset.
The applet developer must highlight in his application the processes longer than 2 sec. and must then accordingly insert MORE TIME proactive commands whenever required.

Example :

```
public class exampleF7 extends Applet
implements ToolkitInterface{

  byte[] transientBuffer;

  public exampleF2() {
    transientBuffer =
    JCSystem.makeTransientByteArray((short)10
    0, JCSystem.CLEAR_ON_RESET);
  }
```

```
  void processToolkit(byte event){
    switch (event) {
      case EVENT_MENU_SELECTION:
        for (short s=0; s<200; s++) {
            doSomeProcessing();
        }
      default:
        break;
    }
```
	The following routine will execute upon menu selection, the method doSomeProcessing() 200 times, thus is likely to take a long time to execute, thus exceeding 2 seconds.

```
  void processToolkit(byte event){
    ProactiveHandler proh;
    switch (event) {
      case EVENT_MENU_SELECTION:
        for (short s=0; s<200; s++) {
          doSomeProcessing();
          if ((short)(s%(short)50) == (short)0)
      {
            proh =
```
	The green routine allows sending a MORE_TIME proactive command every 50 loop-rounds.

```
    ProactiveHandler.getTheHandler;
         proh.init((byte)PRO_CMD_MORE_TIME,
                    (byte)0x00, DEV_ID_ME);
         proh.send();
       }
     }
    default:
       break;
  }
```

Note for the example: in Release 6, a standard method initMoreTime()has been integrated.

F8 - In reentrance, take into account that the OS resources are limited: proactive handlers, available RAM.
Toolkit applications must be tested in reentreance, ie when a CAT context is already active.

Working in reentrance

Description :
Re-entrance refers to the case whereby a proactive session (initiated by an application A) execution is interrupted when a second application B (which can be the same one) is activated. The nested application B (in other words, the application triggered while another application is already activated) has its own file and access conditions context. After application B has been finished, and no additional event occurs before the terminal response is received, control is returned to the first application, so that its own execution can be finished.

In that re-entrance case, the handlers may not be available to the application, so a related exception will be thrown. It will impose the application to manage properly that exception, meaning have a try-catch section to correctly process the application when handlers are not available.

Example :

```
public class exampleF8 extends Applet implements
ToolkitInterface, ToolkitConstants {

  static byte[] text = {(byte)'t', (byte)'e',
  (byte)'x', (byte)'t'};

  void processToolkit(byte event) {
    ProactiveHandler proh;
    switch (event) {
      case EVENT_MENU_SELECTION:
        try {
          proh = ProactiveHandler.getTheHandler();
          proh.initDisplayText((byte)0x80,
          DCS_8_BIT_DATA, text, (short) 0,
          (short) text.length) ;
          proh.send() ;
        }
        catch (ToolkitException te) {
          if (te.getReason() ==
      ToolkitException.HANDLER_NOT_AVAILABLE) {
          }
            // process reentrance mngt here
            …
        }
```

The proactive handler is retrieved in a try-catch clause, thus in case of reentrance, the handler may not be available, and the code in the catch clause will then be executed.

Note: the test of the exception reason is a reinforcement to be sure of the exception type. Depending of the application, this test may be useless.

```
      default:
         break;
   }
}
```

5 Anti-Tearing recommendations

5.1 Reminder on anti-tearing mechanism

The anti-tearing mechanism aims at ensuring correct card integrity and consistency when the card is pulled out of its terminal or after power loss. The card execution is in that case unexpectedly interrupted. The anti-tearing mechanism ensures the detection of such interruptions and the recovery operations during the next card session.

The Java Card™ memory model is based on objects stored by default in persistent memory. Several mechanisms are provided to ensure data integrity and consistency.

5.1.1 Atomicity on objects' field updating

Because objects are stored in NVM, the update of an object field requires several ms. and there is a potential risk of power loss during this operation. If this happen, the content of the object field would be unpredictable.

To solve that issue, the Java Card Virtual Machine specification mandates that object field update is atomic: either entirely performed or not.

This is enforced by the Virtual Machine that automatically detects the update of objects stored in NVM and uses the backup mechanism to perform the update.

Note that, for optimization purposes, it is possible to use RAM cache. In such case, only the cache flush performs the NVM update and uses the backup mechanism.

5.1.2 Transaction services

In some cases (examples given below), it is necessary to perform several memory updates consecutively but the data integrity relies on the fact that these fields are all updated or not.

The Java Card specification defines a "Transaction System" to solve that issue:

- APIs are provided to Applications in order to start, commit or abort a transaction (see detail below)
- A transaction can be started in an APDU command but shall be completed before the end of this command
- If not completed, the system automatically aborts the ongoing transaction before processing the next command
- Only a single transaction can be started at a time.

The Java Card API specification defines the following services:

- Begin transaction: starts a new transaction. All the following persistent memory updates are subject to rollback
- Commit transaction: ends the ongoing transaction: all pending memory update are definitively performed
- Abort transaction: cancel all persistent memory updates

5.1.3 Embedded services using transaction services

The card platform itself, the embedded applications and the embedded services are also subject to use transaction services.

Examples (non exhaustive list) where transaction services are used:

- Allocation of an object may require several update that can not be performed partially without consequences on the entire system integrity
- Application linking and installation requires several memory update that shall be entirely performed to ensure consistency
- Updating a PIN code implies the update of several internal fields requiring consistency
- …

Depending on the layer using the transaction services, it is possible to use either the Java Card API or to use the underlying native (and proprietary) API.

5.1.4 By-passing the on-going transaction may be necessary

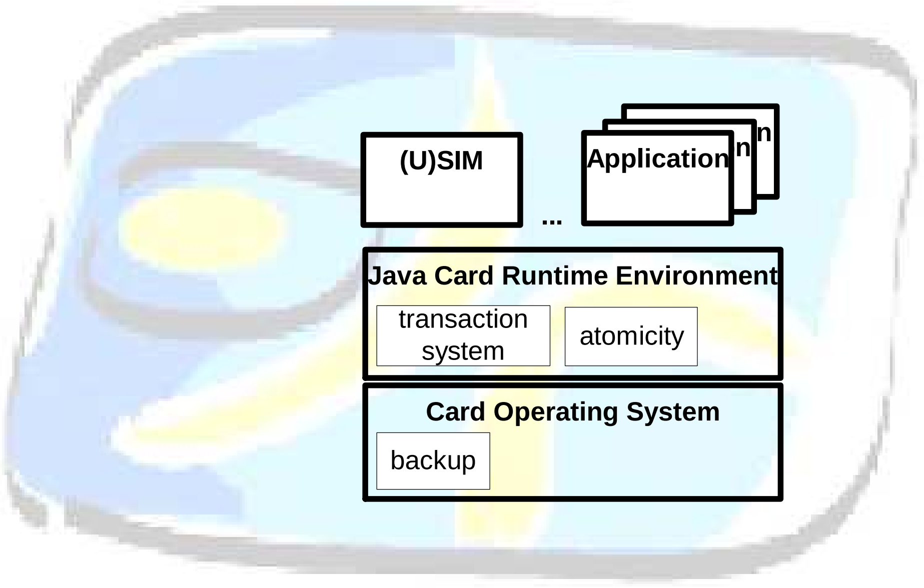
In some circumstances, it may be necessary to by-pass the on-going transaction. Examples:

- It shall not be possible to rollback to the previous value of the ratification counter of a PIN code if its verification failed
- Even if a transaction is in progress, it may be useful to update a large amount of data (i.e an array) without the need to roll-back to its initial value
- The fields update of objects created during the installation of an application does not necessarily require to take part to the on-going transaction because, in case of abortion, the objects will also be discarded
- …

This can be done either using the Java Card API for byte arrays (arrayCopyNonAtomic) or using underlying and proprietary native services.

5.2 Architecture overview

Summarizing, the anti-tearing mechanism is split into different architecture layers.



5.2.1 Level 1 – Backup module

At the lowest level, the backup module, provided by the card operating system, offers services to copy NVM areas from the Heap to a dedicated and secured stack. This module provides standard transaction services: begin, commit and rollback. It is the core engine of the transaction system and follows a specific development process (separated specification and design with re-use in all Gemplus platforms, specific unitary test plan and campaigns, dedicated code-reading procedures and specific security audit).

In some cases, its implementation is adapted in order to take into account NVM characteristics and chip security features. As examples, it may include specific redundancy checksums on backup data, can implement a cyclic transaction stack in order to reduce NVM stress on some pages.

5.2.2 Level 2 – Atomicity and transaction services provided by Java Card

The upper layers do not directly use the backup module. It is encapsulated, as described before, by

- The “atomicity services” directly implemented in the Virtual Machine interpreter
- The “transaction system” accessible via a Java Card API

5.2.3 Level 3 – Use of transaction services by applications

Finally, if the use of atomicity services is implicitly done by applications (each time an object field is updated), the use of the transaction system requires explicit calls to the API whenever required.

The following recommendations give advices and examples on when to use these services.

5.3 Guidelines to protect applications against card tear-out

5.3.1 Optimization of transaction critical section

The **transaction critical section** is the part of the code included between the calls to JCSystem.beginTransaction() and JCSystem.commitTransaction().

T1 – The transaction critical section shall be as small as possible

- **Rationale**
  This rule is standard for critical sections. Both the time and the number of resources implicated in a critical section shall be minimized. For transactions, the aim is to significantly reduce the number of potential errors and also because, by construction, the sequence protected by the transaction is sensitive for the system integrity and consequently for its security and reliability.

- **Example**
  It is necessary to avoid non-necessary calls to other object’s virtual method because there is no control on the side effects like overflow of the transaction buffer, unexpected exception, … The entire reliability may be broken if the method is overwritten later.

```
// WRONG !!
void performDebit(short amount, short dbtID) {
     …
     JCSystem.beginTransaction();
     if( (amount <= this.amount) && isKnownID(dbtID) ){
           this.amount -= amout;
           this.lastDebitID = dbtID;
     }
     JCSystem.commitTransaction();
}

// CORRECT
void performDebit(short amount, short dbtID) {
     …
     if( (amount <= this.amount) && isKnownID(dbtID) ){
           JCSystem.beginTransaction();
           this.amount -= amout;
           this.lastDebitID = dbtID;
           JCSystem.commitTransaction();
     }
     …
}
```

Critical section

T2 – The begin and commit transaction shall be done in the same method

- **Rationale**
  This recommendation is the first necessary step (even if not sufficient) to avoid providing APIs that leave the system with a potential non-committed transaction.
  This design constraint either helps to write small critical sections, helps for code-reading (and audit), and helps to provide strong, scalable and reusable classes.

T3 – All the execution paths of a method starting a transaction shall complete it.

- **Rationale**
  When a transaction is started, all the execution paths shall be studied and shall complete or abort the transaction. This covers conditional jumps but also exception handling.

5.3.2 Protection of transaction critical section

T4 – The transaction critical section shall be protected by a try-catch block

- **Rationale**
  Several exceptions may occur. The first one to take into account is the one thrown if a transaction is already in progress (remind that a single transaction can be started at a time).
  Then, a transaction stack overflow that may occur if the transaction buffer of the underlying platform is not large enough to keep a copy of all modified objects' fields.
  Finally, other exception may also be thrown by methods called within the transaction and shall be handled (see previous recommendation).

- **Example**

```
void performDebit(short amount, short dbtID) {
     if( (amount <= this.amount) && isKnownID(dbtID) ){
     try {
        JCSystem.beginTransaction();
        this.amount -= amout;          // may throw a
TransactionException
        this.lastDebitID = dbtID;      // may throw a
TransactionException
        updateLOGfile(amount,dbtID);   // may throw a UserException
        JCSystem.commitTransaction();
     }
     catch (TransactionException te){
        JCSystem.abortTransaction();
        this.nbErrors++;
     }
     catch (UserException ue){
        JCSystem.abortTransaction();
        this.nbWarning++;
     }
}
```

5.3.3 Protecting objects' allocations

T5 – Allocate object under transaction

- **Rationale**
  Most of the allocations are performed during the application installation, which is automatically protected by a transaction started by the system before calling the install method.
  However, if you need to allocate object later in the code, there is a potential memory leak on cards that does not embed a Garbage Collector.

```
class myApplet extends Applet {
      myObject myField;
      process(...){
           ...
           myField = new myObject();
           ...
     }
}
```

In this example, the object allocation is performed first (execution of byte-code 'new'), then the execution of the object constructor is executed, and finally the storage of its reference into myField is performed.

If a card tear out occurs between the object allocation and the field assignment, the memory allocated by the object will be lost until the application is deleted or the Garbage Collector executed.

The time between the new and the field assignment of the applet is not negligible, even though the developer has the feeling to immediately perform the assignment when writing the Java code.

- **Example**

```
class myApplet extends Applet {
      myObject myField;
      process(...){
           ...
           JCSystem.beginTransaction();
           myField = new myObject();
           JCSystem.commitTransaction();
           ...
     }
}
```

- **Exception**
  If the application targets cards embedding a Garbage Collector (available since Java Card™ 2.2), it is not absolutely necessary to apply this recommendation because the unreachable object will be automatically collected.

5.4 Recommendations for specific and proprietary APIs

5.4.1 Protecting the use of ‘object de-allocation’

T6 – De-allocate objects under transaction

- **Rationale**
  Object de-allocation is provided by a proprietary API and is dedicated for very specific purposes. **Each time possible, the use of interoperable Garbage Collector shall be done instead of using this method**.

  Using this method is symmetrical to allocation and requires the same caution.

```
void process(…){
     …
     if ((myBuffer != null) && (newSize > oldSize)) {
        Gsystem.deAllocate(myBuffer);
        MyBuffer = new byte[newSize];
     }
}
```

If a tear out occurs between the buffer de-allocation and the new buffer creation, the buffer is marked by the system as free whereas the application has kept a reference on that free buffer resulting in a dangling pointer.

In that case, the critical section is made of the de-allocation and the allocation of the new buffer.

- **Example**

```
void process(…){
  …
  if ((myBuffer != null) && (newSize > oldSize)) {
     try {
        JCSystem.beginTransaction();
        Gsystem.deAllocate(myBuffer);
        MyBuffer = new byte[newSize];
        JCSystem.commitTransaction();
     }
     catch (TransactionException te) {
        if(te.getReason() == TRANSACTION_BUFFER_FULL )
           JCSystem.abortTransaction();
     }
  }
}
```

6 Application development Checklist

Type	Title	Description	Check
Minimizing field issue risks	F1 - stress E2PROM : objects in E2PROM	F1 - Check where all your variables are stored and accordingly move them if there is a risk of NVM stress (guaranteed 100,000 updates for E^{2}PROM and FLASH technologies)	☐
	F2 – event status	F2 - Avoid writing in EEPROM upon STATUS event	☐
	F3 - file update	F3 - Avoid writing in EEPROM upon file update events when the concerned file is a HIGH update activity file	☐
	F4 - Warning functions calls / Jstack	F4.1 - Avoid passing too many parameters in a function (4 or 5 max) F4.2 - Avoid declaring too many local variables inside each method F4.3 - Reduce the overall nesting level (ie a function calling a function). Latest Gemalto Java cards allow more than 20 nesting levels.	☐
	F5 - Objects creation (card with garbage collector or objects are created at install only)	F5 - ALL objects making up the application are created during installation.	☐
	F6 - Transactions (ex : Begin and Abort mngt with exception, …)	F6 - If your application uses the Java Card transactions: • The transaction critical section (ie part between a Begin and a Commit) shall be as small as possible. • The begin and commit shall be done in the same method. • All execution paths of a method starting a transaction shall complete it. • The transaction critical section shall be protected by a try-catch block	☐
	F7 - Verification that the execution time is < 2s : More Time / Trig	F7 - Insert a MORE TIME proactive command if your toolkit process is longer than 2s in order not to block the handset	☐

	applet		
	F8 - Working in reentrance	F8 - In re-entrance, take into account that the OS resources are limited : proactive handlers, available RAM. Toolkit applications must be tested in re-entrance, ie when a CAT context is already active.	☐
Development basics	D1 – Use APIs	Use APIs, rather than rewriting methods, whenever possible	☐
	D2 – Coding PINs & Keys	Coding and storing PINs and Keys in primitive arrays must be avoided	☐
	D3 – Sensitive data – initializations & clearing	Sensitive data must be initialized at the beginning and clears at the end of the session	☐
	D4 – Sensitive data - Storage	Store your session data in transient data. Avoid using the APDU buffer to store these data	☐
	D5 - Protect your sensitive data against Rollback attack	Counter management must be implemented to combat against Rollback attacks when working with sensitive data.	☐
	D6 - Initialized buffers declared as static	All initialized buffers (containing menu item, strings…) must be declared as static:	☐
	D7 - reentrance	How to deal with re-entrance? • Requiring an SMS-Submit • Throwing an ISOException • Event Proactive Handler Available	☐
	D8 - handlers availability	See example above	☐
	D9- Global vs local variables	Check if you use the right variable type: • **Instance variables (global variables):** created at the object instantiation. • **Local variables**: they are accessible only from the function or block in which it is declared.	☐
	D10 - Storage of references to temporary entry point objects is	the storage of references should be done throw a local reference	☐

	forbidden		
	D11 – Keep methods sizes small	Break down complex methods into smaller methods	☐
	D12 – Ensure methods do not exceed 256 methods per class	Ensure that number of methods per class do not exceeded 256	☐
Anti Tearing	T1 – The transaction critical section shall be as small as possible	Time and no of resources in critical section are to be kept to the minimum possible	☐
	T2 – The begin and commit transaction shall be done in the same method	Check that the begin and commit transaction is within the same method	☐
	T3 – All the execution paths of a method starting a transaction shall complete it.	Check that all execution paths are completed or aborted. This should include error or exception handling routines	☐
	T4 – The transaction critical section shall be protected by a try-catch block	To prevent transaction stack overflow during an exception, the transaction critical section must be protected using a try-catch block	☐
	T5 – Allocate object under transaction	If objects are allocated later in the code and not during application installation, check that your object creation is done within transaction. If Garbage collector is activated (available since Java Card™ 2.2), this is not mandatory.	☐
	T6 – De-allocate objects under transaction	Deallocate objects method is used only when a tear out is detected by the method. For all other time, the use of interoperable Garbage Collector shall be done instead of using this method	☐

END OF DOCUMENT