GLOBALPLATFORM®
THE STANDARD FOR SECURE DIGITAL SERVICES AND DEVICES

# GlobalPlatform Technology

# Executable Load File Upgrade

## Card Specification v2.3 – Amendment H

## Version 1.1

**Public Release**

**March 2018**

**Document Reference: GPC_SPE_120**

THIS SPECIFICATION OR OTHER WORK PRODUCT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE COMPANY, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT.

# Contents

# Tables

# 1 Introduction

This document defines an extension of the GlobalPlatform Card Specification ([GPCS]) to facilitate the upgrade of Executable Load Files (ELFs) present in a Secure Element.

## 1.1 Audience

This amendment is intended primarily for card manufacturers and application developers developing GlobalPlatform card implementations.

It is assumed that the reader is familiar with smart cards, smart card production, [GPCS], and [JavaCard].

## 1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://www.globalplatform.org/specificationsipdisclaimers.asp. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

## 1.3 References

**Table 1-1: Normative References**

| Standard / Specification | Description | Ref |
|---|---|---|
| GlobalPlatform Card Specification | GlobalPlatform Card Specification v2.3.1 | [GPCS] |
| ETSI TS 102 226 | Smart cards; Remote APDU structure for UICC based applications, European Telecommunications Standards Institute Technical Committee Smart Card Platform (TC SCP) | [TS 102 226] |

**Table 1-2: Informative References**

| Standard / Specification | Description | Ref |
|---|---|---|
| Java Card™ | http://www.oracle.com/technetwork/java/embedded/javacard | [JavaCard] |

## 1.4 Terminology and Definitions

Terms used in this document are defined in [GPCS].

## 1.5 Abbreviations and Notations

Abbreviations and notations used in this document are included in Table 1-3.

**Table 1-3: Abbreviations and Notations**

| Abbreviation / Notation | Meaning |
|---|---|
| AID | Application Identifier |
| APDU | Application Protocol Data Unit |
| API | Application Programming Interface |
| BIP | Bearer Independent Protocol |
| CAT_TP | Card Application Toolkit Transport Protocol |
| CLA | CLAss byte of the command message |
| C-MAC | Command MAC |
| DAP | Data Authentication Pattern |
| INS | INStruction byte of the command message |
| Lc | Exact length of data in a case 3 or case 4 command |
| Le | Maximum length of data expected in response to a case 2 or case 4 command |
| LV | Length Value |
| MAC | Message Authentication Code |
| NVRAM | Non-Volatile Random Access Memory |
| P1 | Reference control Parameter 1 |
| P2 | Reference control Parameter 2 |
| RAM | Random Access Memory |
| RAS | Remote Administration Server |
| RFU | Reserved for Future Use |
| SD | Security Domain |
| SIM | Subscriber Identity Module |
| SIO | Shareable Interface Object |
| STK | SIM Tool Kit |
| SW1 | Status Word One |
| SW2 | Status Word Two |
| TCP | Transmission Control Protocol |
| TLV | Tag Length Value |
| Var. | Variable |

## 1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n.n*.1, *n.n*.2, etc., are maintenance releases that incorporate errata and precisions; all non-trivial revisions are indicated, often with revision marks.

**Table 1-4: Revision History**

| Date | Version | Description |
|---|---|---|
| February 2017 | 1.0 | Initial Public Release |
| March 2018 | 1.1 | Added optional support for the upgrade of multiple ELFs within the same ELF Upgrade Session. If supported, Library ELFs may also be specified as part of upgraded ELFs.<br>Removed Upgrade Option “Preserve old ELF for Recovery Procedure”.<br>Added new P1 option for MANAGE ELF UPGRADE command to forcefully start the Recovery Procedure when still in the Loading Phase.<br>Added a few clarifications here and there. Fixed some examples in Annex A.<br>Added new informative Annex C “Remote Administration”. |

# 2 Use Case and Requirements

Today, all modern digital devices (laptops, handsets, etc.) allow for a software upgrade process, especially where there is a network connection available. The reasons for upgrading software are numerous: fixing bugs, adding new features, etc. With billions of pieces deployed, smart cards definitely need their own software upgrade mechanism.

Regarding deployment, the fundamental difference between upgrading classic software and smart card software is the existence of persistent instances of software programs. These instances are created and personalized by Service Providers with both user data and secret or sensitive data. The personalization process can sometimes be complex and expensive and ideally should not be repeated in case of software upgrades. It is not a good solution for Service Providers to simply delete their Applications (and related data), then install and personalize them again.

Another aspect of service deployment on smart cards is that an ELF may be shared and used by several Service Providers (e.g. a mobile payment application, which may be instantiated by different banks). Hence, updating an ELF is not only (or not at all) the business of a single Service Provider, but rather is the business of the ELF provider.

Finally, it is necessary to consider some of the technical characteristics of smart cards, which in general:

- May be rather limited in memory space (both RAM and NVRAM). It is very common that after personalization and deployment, a smart card has almost no memory space left.
- Are required to recover from power loss in a consistent manner, as much as possible.

This document focuses on GlobalPlatform cards implementing the Java Card Specifications (see [JavaCard]). In particular, it shall be understood that:

- An Executable Load File is a Java Card package.
- An Executable Module is a Java Card Applet class.
- An Application is a Java Card Applet instance.

Each of these will be identified by an AID (Application Identifier).

# 3 Executable Load File Upgrade

This chapter describes the Executable Load File Upgrade Process. The ELF being upgraded will be commonly referred to as the “old ELF version” and the ELF upgrading the old ELF version will be commonly referred to as the “new ELF version”. The ELF Upgrade Process is basically composed of the following phases:

- Saving Phase

  Each existing Application instance created from the old ELF version can save its instance data using the Upgrade API. Registry data (instance data that is not manipulated directly by Application instances) is saved automatically. The old ELF version, its Application instances, and their instance data are then deleted, except for the instance data saved using the Upgrade API and the automatically saved Registry data. The Loading Phase can then be started immediately or be postponed and performed later.

- Loading Phase

  During this phase, the new ELF version is loaded. Before loading the new ELF version, library ELFs previously imported by and left unused after the deletion of the old ELF version may be deleted/replaced, unrelated ELFs may be deleted/loaded, and new library ELFs may be loaded (e.g. a new library ELF imported by the new ELF version). After the new ELF version has been loaded, the Restore Phase can start.

- Restore Phase

  New Application instances are automatically created from the new ELF version, in the same number and with the same AIDs as previously existing Application instances. Each new Application instance is permitted to restore the instance data from the previous Application instance that had the same AID. The Registry data of the previous Application instance is also automatically restored and associated with the new Application instance.

The following variants are also described in the document:

- Loading the new ELF version before starting the ELF Upgrade Session. This makes it possible to chain the different phases automatically.
- Optional capability of upgrading more than one ELF within the same ELF Upgrade Session.

## 3.1 General Requirements

The implementation shall enforce the following requirements:

- Only a single ELF Upgrade Session shall be processed at a time. No new ELF Upgrade Session may be started until the previous one (if any) has completed or aborted.
- The implementation may support upgrading more than one ELF during an ELF Upgrade Session.

## 3.2 Runtime Processing Requirements

### 3.2.1 Upgrade Session Start

The ELF Upgrade Session starts upon receipt of a MANAGE ELF UPGRADE [start] command which triggers preliminary checks and the Saving Phase.

The MANAGE ELF UPGRADE [start] command shall be rejected with an error and the ELF Upgrade Process shall be aborted if any of the following conditions is satisfied:

- An ELF Upgrade Session is already ongoing.
- The command specifies a minimum version number and the version number of the old ELF version being upgraded is lower than the specified minimum.
- Static dependencies exist that would prevent the old ELF version from being deleted (according to Java Card rules).
- Any Application instance created from the old ELF version is currently selected (on any logical channel) or suspended (according to STK rules).
- The new ELF version is already present on the card (i.e. with an AID other than the old ELF version) and:
  - The new ELF version does not conform to the requirements described in section 3.2.4.1; or
  - Application instances have already been created from the new ELF version.

If multiple ELFs must be upgraded within the same ELF Upgrade Session, the session shall start with multiple MANAGE ELF UPGRADE [start] commands, each one specifying an ELF that shall be upgraded, and the above checks shall be performed for each of these ELFs. See section 4.1 for more details.

Once the ELF Upgrade Session has been started (and until it is completed or aborted), the following rules shall apply:

- Any unrelated operation involving an Application instance or ELF that is the subject of the ELF Upgrade Process shall be rejected. For example:
  - Selection of an Application instance (or triggering of an STK Application instance) involved in the ELF Upgrade Process
  - Deletion of an Application instance or ELF involved in the ELF Upgrade Process
  - Installation of a new Application instance from an ELF involved in the ELF Upgrade Process
  - Obtaining a Shareable Interface Object from an Application instance involved in the ELF Upgrade Process
- It shall not be possible to delete a Security Domain that is associated with any of the ELFs and/or Application instances involved in the ELF Upgrade Process (as it shall be possible to associate the new ELF versions and/or new Application instances with that same Security Domain). Other unrelated Security Domains may still be deleted.

The ELF Upgrade Session first enters the Saving Phase.

### 3.2.2 Saving Phase

The Saving Phase is composed of the following sequences:

- Data Saving Sequence
- Cleanup Sequence
- Deletion Sequence

If multiple ELFs are upgraded within the same ELF Upgrade Session, each of these sequences shall be completed for all the ELFs before entering the next sequence.

First, the Saving Phase enters the Data Saving Sequence. All Application instances created from the ELF(s) being upgraded shall be requested by the OPEN to save their instance data:

- The OPEN shall invoke the `onSave()` method of each Application instance, in the same order the Application instances were initially installed (i.e. historical order). If an Application instance does not implement the `OnUpgradeListener` interface, then the OPEN simply moves to the next Application instance.
- In the `onSave()` method, the Application instance uses the Upgrade API to save the instance data it wishes to migrate toward a new Application instance that will be created from the new ELF version.
  - As the goal is to migrate instance data to a new Application instance, the implementation of this method shall not perform any modification of the instance data, nor cause any unnecessary side effect. Following this requirement also guarantees that the Application instance will recover its initial state in case the Data Saving Sequence is aborted (see abortion cases below). The Application instance shall wait for the execution of its `onCleanup()` method (see Cleanup Sequence below) to perform any cleanup operation required to facilitate Application instance deletion.
  - If this method returns with no error, the OPEN shall automatically save the Registry Data of the Application instance. The exact content of Registry Data depends on the type of card implementation, but it generally includes any persistent on-card information relating to the Application instance that would not be stored/manipulated directly by the Application instance itself.
- Using the Upgrade API, it is possible to save any object except a custom object. A custom object is an instance of a custom class, i.e. a class belonging to any of the packages currently being upgraded. Custom objects cannot be saved directly because the class hierarchies of the new package versions may be quite different, so that custom objects might no longer make sense in the new package versions (i.e. it would not be possible to restore objects created from classes that no longer exist). Nevertheless, the Upgrade API provides means to deal with custom objects by transferring their content to "generic objects" (`Element`) that can be saved. Attempts to save custom objects during the Saving Phase shall be detected by the end of the Cleanup Sequence, and shall result in the automatic abortion of the ELF Upgrade Process. In this case, a specific status word shall be returned to clearly indicate the reason for the abortion.
- Saving objects may create so-called "residual" dynamic dependencies:
  - The OPEN shall not delete an object that was saved during the Saving Phase. For example, a Shareable Interface Object (SIO) provided by a server Application may have been saved. In this case, if the SIO is released by the server Application and the Java Card Garbage Collector is invoked, this SIO shall not be collected/deleted.

  - If an Application instance A from package P which is being upgraded saves an object O created from a class defined in package Q not being upgraded, then a dynamic dependency toward Q will still exist internally. This dynamic dependency will prevent Q from being deleted (e.g. during the Loading Phase) even after A and P have been deleted, because Q will be needed at the time object O is restored. During the ELF Upgrade Process (i.e. as long as the ELF Upgrade Process hasn't completed or aborted), the OPEN shall check the existence of such residual dynamic dependencies before accepting the deletion (requested by a DELETE command) of an ELF or Application.

- An Application instance requested to save its instance data shall assume that the execution of its `onSave()` method may be interrupted (e.g. power loss). In case of interruption, it shall be possible to execute the `onSave()` method again safely (e.g. with no exception).
- If the `onSave()` method throws an exception, the Data Saving Sequence and the entire ELF Upgrade Process shall be aborted (i.e. this is a fatal error).
- The Data Saving Sequence is completed once all `onSave()` methods have been invoked successfully.

Note that the contents of static fields will not be saved automatically. These will need to be saved explicitly, similarly to instance fields. Application instances are responsible for saving such contents in their `onSave()` method for later restoration during the Restore Phase.

Next, the Saving Phase enters the Cleanup Sequence. The OPEN requests all Application instances created from the old ELF version(s) to perform any cleanup operation required to facilitate deletion:

- If an Application instance does not implement the `OnUpgradeListener` interface, then the OPEN simply moves to the next Application instance. The OPEN shall invoke the `onCleanup()` method of each Application instance. Any exception thrown by this method shall be caught and ignored.
- An Application instance requested to perform cleanup operations shall assume that the execution of its `onCleanup()` method may be interrupted (e.g. power loss). In case of interruption, it shall be possible to execute the `onCleanup()` method again safely.
- Note that the `Applet.uninstall()` method will not be invoked during the Saving Phase. The `uninstall()` method would typically undertake more drastic actions (e.g. clearing key objects, etc.) required before definitively deleting the Application instance from the card. The goal of the `onCleanup()` method is not to clear objects (which must actually be preserved and migrated) but only to facilitate the deletion before replacement (e.g. by releasing some dynamic dependencies).
- The Cleanup Sequence is completed once all `onCleanup()` methods have been invoked successfully.

Finally, the Saving Phase enters the Deletion Sequence. The OPEN shall attempt to delete the ELF(s) and all its (their) Application instances:

- The OPEN shall not invoke the `Applet.uninstall()` method.
- The OPEN shall check that the ELF(s) and all its (their) Application instances can be deleted together. If the deletion is rejected, then the Saving Phase shall be aborted.
- A special deletion algorithm shall apply during this phase:
  - Application instances shall be removed from the OPEN Registry; however (as previously mentioned), their Registry Data shall be preserved for reuse during the Restore Phase.
  - Only custom objects (including the Application instance itself) and non-custom objects that were not saved using the Upgrade API shall be deleted. Non-custom objects saved using the Upgrade API shall not be deleted and shall be preserved for reuse during the Restore Phase.

The Saving Phase successfully completes when all the Application instances and ELF(s) are deleted. If the Saving Phase is interrupted by a power loss, then:

- If the deletion of the Application instances and ELF(s) (atomic and irreversible operation) has already started, then it shall automatically restart and complete upon next power up. Upon completion of the Deletion Sequence, the process enters the Loading Phase.
- A MANAGE ELF UPGRADE [status] command may be sent to retrieve the status of the ELF Upgrade Session.
- A MANAGE ELF UPGRADE [resume] command may be sent to resume the Saving Phase.
  - If the interruption occurred during the Data Saving Sequence, the OPEN shall invoke the Java Card Garbage Collector for the last Application instance whose `onSave()` method was invoked and could not return (to collect working data potentially created and lost) and then restart, invoking `onSave()` methods starting from that Application instance.
  - If the interruption occurred during the Cleanup Sequence, the OPEN shall restart, invoking `onCleanup()` methods from the last one that could not return.
  - If the interruption occurred during the Deletion Sequence and the latter did not complete automatically (i.e. the irreversible deletion operation did not start already), the Deletion Sequence shall restart.
- The MANAGE ELF UPGRADE [abort] command may be sent to abort the Saving Phase. All session data shall be deleted. The ELF(s) and its (their) Application instances shall be left in their current states. The Java Card Garbage Collector shall be invoked for all Application instances.
  - If the interruption occurred during the Data Saving Sequence and `onSave()` methods were implemented with no side effect, this should leave Application instances in the state they were in before the start of the ELF Upgrade Process.
  - If the interruption occurred during the Cleanup Sequence or the Deletion Sequence (i.e. actual deletion did not start), Application instances may be in an incorrect state (depending on the operations performed by `onCleanup()` methods).

After the Saving Phase is completed, the process enters the Loading Phase.

### 3.2.3 Loading Phase

The off-card entity may have loaded the new ELF version(s) on the card before the start of the ELF Upgrade Session. Note that this could only occur if the old and new ELF versions do not share the same AID(s).

If the new ELF version(s) is (are) not already present on the card, then the off-card entity shall load the new ELF version(s). Before doing so, the off-card entity may delete or load other Executable Load Files (e.g. replace or load libraries needed by the new ELF version(s)).

The loading of the Executable Load Files shall be performed according to the usual rules. The INSTALL [for load] command shall be used to start the loading and LOAD commands shall be used to transmit the Load File. DAP Blocks, if present, shall be verified. See section 3.4 for restrictions applicable to loading operations during the ELF Upgrade Session.

Upon successful loading of the new ELF version(s):

- If session options indicate that the Restore Phase shall start automatically (see session option bit b3 in Table 4-4), then the ELF Upgrade Process shall enter the Restore Phase directly at the end of the loading process. In this case, the response to the last LOAD command shall not be returned immediately, but shall only be returned upon completion or failure of the ELF Upgrade Process. In case of failure, the LOAD command shall return one of the status words described in Table 4-10, MANAGE ELF UPGRADE Error Conditions. If multiple ELFs are being upgraded, then multiple ELFs are expected to be loaded, and the Restore Phase shall only start after all expected ELFs have been detected on the card (i.e. on receipt of the last LOAD command of the last ELF).
- Otherwise, the response to the last LOAD command shall be returned as usual, and the ELF Upgrade Process shall wait for a MANAGE ELF UPGRADE [resume] command to start the Restore Phase.

At any time during the Loading Phase:

- A MANAGE ELF UPGRADE [status] command may be sent to retrieve the status of the ELF Upgrade Session.
- A MANAGE ELF UPGRADE [resume] command may be sent. If the new ELF version(s) is (are) not present yet, nothing is done and the response simply indicates the status of the ELF Upgrade Session. If the new ELF version(s) is (are) present, the Restore Phase shall start.
- A MANAGE ELF UPGRADE [recovery] command may be sent to forcefully start the ELF Upgrade Recovery Procedure (see section 3.2.4.2) even if the Restore Phase hasn't been entered.
- A MANAGE ELF UPGRADE [abort] command may be sent to abort the Loading Phase and the entire ELF Upgrade Process. In this case, all session data and saved instance data shall be deleted.

### 3.2.4 Restore Phase

The Restore Phase is composed of the following sequences:

- Installation Sequence
- Restore Sequence
- Consolidation Sequence

If multiple ELFs are upgraded within the same ELF Upgrade Session, each of these sequences shall be completed for all the ELFs before entering the next sequence.

First, the Restore Phase enters the Installation Sequence:

- The OPEN shall check whether the new ELF version(s) complies(y) with the requirements described in section 3.2.4.1. If not, the Restore Phase shall be aborted and the recovery procedure described in section 3.2.4.2 shall be started.
- The OPEN shall automatically create new Application instances from the new ELF version(s), in the same number and with the same AID as previously existing Application instances. For each new Application instance that needs to be created, the required Application module shall be located in the new ELF version and the corresponding `Applet.install()` method shall be invoked:
  - The OPEN is responsible for ensuring that the parameters `(bArray, bOffset, bLength)` passed to the `install()` method contain the following information:
    - The `bArray` byte array shall contain the following consecutive LV coded data:
      - Length of Instance AID
      - Instance AID, which shall be the instance AID of one of the Application instances being upgraded
    - The `bOffset` parameter shall indicate the offset within `bArray`, pointing to the length of the instance AID.
    - The `bLength` parameter shall indicate the total length of the data defined above.
  - In its `install()` method, the Application instance:
    - Shall ensure that it is not invoked in the context of a regular installation but in the context of an upgrade by calling the `UpgradeManager.isUpgrading()` method.
    - Shall register with the Instance AID specified in Install Parameters. In the absence of regular Application Specific Parameters, it may also perform basic initializations; however, it is the role of the `onRestore()` method (invoked later; see below) to restore saved instance data and perform all necessary initializations.
  - Upon registration of the new Application instance, the OPEN shall:
    - Associate the Application instance to the same Security Domain as before.
    - Assign the Application instance the same privileges as before (if any).
    - Restore the Registry Data of the Application instance.
  - If, for any reason, the installation of an Application instance fails, the Installation Sequence and the entire Restore Phase shall be aborted, and the recovery procedure described in section 3.2.4.2 shall be started.
  - The Installation Sequence is completed once all new Application instances have been installed.

Next, the Restore Phase enters the Restore Sequence. The OPEN shall request each new Application instance to restore its instance data:

- The OPEN shall invoke the `onRestore()` method of each Application instance in the same order as it invoked the `onSave()` method for previous Application instances. If an Application instance does not implement the `OnUpgradeListener` interface, then the OPEN simply moves to the next Application instance.
- In the `onRestore()` method, the Application instance:
  - Shall use the Upgrade API to restore instance data. The link between suitable instance data is ensured by the API (i.e. per AID). In particular, the ownership of restored objects that originally belonged to the Application instances being upgraded is automatically transferred to the new Application instances. The ownership of other objects shall remain unchanged.
  - May use the `UpgradeManager.checkPreviousPackageAID()` method to check the AID of the old ELF version and determine different restore strategies.
  - May use the `UpgradeManager.getPreviousPackageVersion()` method to check the old ELF version number and determine different restore strategies.
  - May initialize new fields and data structures, if any.
- An Application instance requested to restore instance data shall assume that the execution of its `onRestore()` method may be interrupted (e.g. power loss). In case of interruption, it shall be possible to execute the `onRestore()` method again safely (e.g. with no exception).
- If the `onRestore()` method throws an exception, the Restore Sequence and the entire Restore Phase shall be aborted, and the recovery procedure described in section 3.2.4.2 shall be started.
- The Restore Phase is completed when all `onRestore()` methods have been invoked successfully. All session data shall be deleted. The Java Card Garbage Collector shall be invoked for all Application instances, to collect any possible working data and non-restored (and now non-reachable) instance data.

Finally, the Restore Phase enters the Consolidation Phase. The OPEN shall request each new Application instance to perform any operation required to consolidate its data:

- The OPEN shall invoke the `onConsolidate()` method of each Application instance. Any exception thrown by this method shall be caught and ignored. If an Application instance does not implement the `OnUpgradeListener` interface, then the OPEN simply moves to the next Application instance.
- An Application instance requested to consolidate its data shall assume that the execution of its `onConsolidate()` method may be interrupted (e.g. power loss). In case of interruption, it shall be possible to execute the `onConsolidate()` method again safely (e.g. with no exception).
- The Consolidation Sequence (and Restore Phase) is completed once all `onConsolidate()` methods have been invoked successfully.

If the Restore Phase is interrupted by a power loss, then:

- A MANAGE ELF UPGRADE [status] command may be sent to retrieve the status of the ELF Upgrade Session.
- A MANAGE ELF UPGRADE [resume] command may be sent to resume the Restore Phase.
  - If the interruption occurred during the Installation Sequence, the OPEN shall restart installing new Application instances from the last one that failed to install.
  - If the interruption occurred during the Restore Sequence, the OPEN shall restart invoking the `onRestore()` method from the last one that could not return.

- If the interruption occurred during the Consolidation Sequence, the OPEN shall restart invoking the `onConsolidate()` method from the last one that could not return.

- A MANAGE ELF UPGRADE [abort] command may be sent to abort the Restore Phase and the entire ELF Upgrade Process. All session data, saved instance data, new Application instances, and the new ELF version(s) shall be deleted.

#### 3.2.4.1 Requirements for the New ELF Version(s)

The following requirements apply to the new ELF version(s) (i.e. replacing the old one(s)):

- Each new ELF version shall be associated with the same Security Domain as the old ELF version.
- Each new ELF version shall define at least the same Application modules as the old ELF version, with the exact same AIDs. New Executable Modules may also be present. The purpose of this requirement is to establish a link between old and new Application instances. Before proceeding, the OPEN shall check that all required module AIDs are defined in the new ELF version.

  NOTE: [GPCS] section 6.5.1.1 says "Each Executable Load File or Executable Module is associated with an AID that shall be unique on the card." This statement is partly superseded by this amendment, as the same Executable Module AIDs shall be defined in the old and new ELF versions.

#### 3.2.4.2 ELF Upgrade Recovery Procedure

If the Restore Phase fails due to error detection, the ELF Upgrade Session shall not be fully aborted but shall enter the ELF Upgrade Recovery Procedure, which is composed of the following steps:

- All the new Application instances and new ELF version(s) shall be deleted. The same special deletion algorithm used during the Saving Phase shall be used; that is, non-custom objects saved using the Upgrade API during the Saving Phase shall not be deleted and shall be preserved.
- The upgrade session shall go back to the Loading Phase with a special session status of WAITING_RESTORE_FAILED (see section 3.3).
- The off-card entity shall load the old ELF version(s) again. The implementation shall check that the AID(s) and version number(s) of the loaded ELF(s) match the ones of the old ELF version(s). If the old ELF version(s) contain(s) suitable restore code, then the ELF Upgrade Process should be able to recover the initial state prior to the start of the ELF Upgrade Session.

Notice that the success of this procedure depends on the ability of the old ELF version(s) to restore the data it (they) previously saved during the Saving Phase.

Notice also that, at any time, the off-card entity may choose to abort the entire ELF Upgrade Session with a MANAGE ELF UPGRADE [abort] command.

Finally, the ELF Upgrade Recovery Procedure may only apply once per ELF Upgrade Session. That is, if the Restore Phase launched as part of the recovery procedure (with the old ELF version(s)) also fails, then the ELF Upgrade Session shall be fully aborted.

## 3.3 ELF Upgrade Session Status

At any time, it shall be possible to send a MANAGE ELF UPGRADE [status] command to query information about an ELF Upgrade Session. The following list describes the possible session status:

- NO_UPGRADE_SESSION: There is no upgrade session ongoing.
- WAITING_EXECUTABLE_LOAD_FILE: The upgrade session is in the Loading Phase and is waiting for the new ELF version(s) to be loaded. This session state will be skipped if expected new ELF version(s) was (were) loaded prior to the start of the upgrade session.
- WAITING_RESTORE: The new ELF version(s) has (have) been detected on the card. The upgrade session is in the Loading Phase and is waiting for a MANAGE ELF UPGRADE [resume] command to transition from the Loading Phase to the Restore Phase. This session state will be skipped if session options specify automatic transition to the Restore Phase.
- WAITING_RESTORE_FAILED: A failure occurred during the Restore Phase. The new ELF version(s) has (have) been deleted and the Recovery Procedure (see section 3.2.4.2) has been started. Notice that if the loading of the old ELF version(s) is required, started, and interrupted again, the session status will be WAITING_EXECUTABLE_LOAD_FILE again, but the expected ELF version(s) will be the old ELF version(s).
- INTERRUPTED: The upgrade session is temporarily interrupted and waiting for a MANAGE ELF UPGRADE [resume] or [abort] command. Sub-states are:
  - INTERRUPTED_SAVING: A power loss occurred in the Data Saving Sequence (Saving Phase).
  - INTERRUPTED_CLEANUP: A power loss occurred in the Cleanup Sequence (Saving Phase).
  - INTERRUPTED_DELETE: A power loss occurred in the Delete Sequence (Saving Phase).
  - INTERRUPTED_INSTALL: A power loss occurred in the Installation Sequence (Restore Phase).
  - INTERRUPTED_RESTORE: A power loss occurred in the Restore Sequence (Restore Phase).
  - INTERRUPTED_CONSOLIDATE: A power loss occurred in the Consolidation Sequence (Restore Phase).
- UPGRADE_COMPLETED: The upgrade session has successfully completed or the Recovery Procedure (see section 3.2.4.2) has successfully completed (NOTE: a warning is returned in this case).

The coding of the ELF Upgrade Session Status is described in Table 4-8.

## 3.4 Card Content Management During ELF Upgrade Session

Except as listed below, Card Content Management operations may be performed as usual during an ELF Upgrade Session (i.e. between a MANAGE ELF UPGRADE [start] command and the full completion of the session, successful or not). This behavior allows other operations not related to the ELF Upgrade Session to be performed during the Loading Phase, or when the ELF Upgrade Session was interrupted by a power loss and (after a card reset) is waiting for a MANAGE ELF UPGRADE [resume] command.

The following operations shall always be rejected during an ELF Upgrade Session:

- DELETE command attempting to delete an Application instance created from either the old or new ELF version(s)
- DELETE command attempting to delete either the old or new ELF version(s)
- Attempt to reload (any of) the old ELF version(s), unless the ELF Upgrade Session has entered the WAITING_RESTORE_FAILED session state (described in section 3.3) and the old ELF version(s) is (are) indeed expected to be reloaded. If the old and new ELF version(s) share the same AID(s), then they shall be further distinguished based on their version numbers (transmitted during the loading operation).
- INSTALL [for load] command attempting to load an ELF with the same AID as one of the Application instances expected to be created from the new ELF version(s) during the Restore Phase
- INSTALL [for install] or INSTALL [for install and make selectable] command attempting to:
  - Install an Application instance from the old ELF version(s) (in case of interruption during the Saving Phase) or from the new ELF version(s) (if present on the card)
  - Install an Application instance with the same AID as one of the Application instances expected to be created from the new ELF version(s) during the Restore Phase
  - Install an Application instance with one of the following privileges if it shall be assigned to one of the Application instances expected to be created from the new ELF version(s) during the Restore Phase:
    - Card Reset
    - Final Application
- INSTALL [for extradition], INSTALL [for registry update], or INSTALL [for personalization] command with an Application AID field identifying one of the Application instances expected to be created from the new ELF version(s) during the Restore Phase

## 3.5 Card Capability Information

Card Capability Information provides details on the optional mechanisms that are actually supported by the card. (See [GPCS] sections 7.4.1.4 and H.4.) This section defines additional information to be included in Card Capability Information when the card supports the ELF Upgrade Process.

The following table amends Table H-5 of [GPCS].

**Table 3-1: Card Capability Information (Amending [GPCS] Table H-5)**

| Tag | Length | Data / Description | Presence |
|---|---|---|---|
| '67' | Variable | Card Capability Information | Mandatory |
| … | | … | |
| '89' | Variable | Supported version of ELF Upgrade Process & Options (see Table 3-2) | Conditional |

Tag '89' shall be present if the card supports the ELF Upgrade Process as described in this document, and shall not be present otherwise. The first byte shall have a value of '01' indicating support for the process and mechanisms described in this version of the document. Subsequent bytes shall encode options supported by implementation. In this version of the document, only a single byte of options is defined (see Table 3-2).

**Table 3-2: ELF Upgrade Process Options**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 0 | Only a single ELF may be upgraded within an ELF Upgrade Session |
| - | - | - | - | - | - | - | 1 | Several ELFs may be upgraded within the same ELF Upgrade Session |
| x | x | x | x | x | x | x | x | RFU |

# 4 APDU Commands

This chapter describes the APDU commands that shall be supported by Security Domains to enable the ELF Upgrade Process.

## 4.1 MANAGE ELF UPGRADE

### 4.1.1 Definition and Scope

The MANAGE ELF UPGRADE command allows starting, resuming, or aborting an ELF Upgrade Session.

This command may only be issued within a Secure Channel Session. The security level for the command depends on the security level defined in the EXTERNAL AUTHENTICATE command.

### 4.1.2 Command Message

The MANAGE ELF UPGRADE command is coded according to the following table.

**Table 4-1: MANAGE ELF UPGRADE Command Message**

| Code | Value | Meaning |
|---|---|---|
| CLA | '80' - '8F', 'C0' - 'CF', or 'E0' - 'EF' | See [GPCS] section 11.1.4 |
| INS | 'EA' | MANAGE ELF UPGRADE |
| P1 | 'xx' | Reference control parameter P1 |
| P2 | 'xx' | Reference control parameter P2 |
| Lc | 'xx' | Length of data field |
| Data | 'xxxxx…' | Upgrade Session Data (and C-MAC if present) |
| Le | '00' | |

#### 4.1.2.1 Reference Control Parameter P1

The reference control parameter P1 defines the specific role of the MANAGE ELF UPGRADE command and also allows for command data to be longer than 255 bytes and to be segmented into arbitrary components and transmitted in a series of MANAGE ELF UPGRADE commands. It is coded according to the following table.

**Table 4-2: MANAGE ELF UPGRADE Command Reference Control Parameter P1**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|
| 0 | - | - | - | - | - | - | - | Last (or only) command |
| 1 | - | - | - | - | - | - | - | More commands |
| - | - | - | 0 | 0 | 0 | 0 | 1 | Start New Upgrade Session – Last (or only) ELF Upgrade Request |
| - | - | - | 1 | 0 | 0 | 0 | 1 | Start New Upgrade Session – More ELF Upgrade Requests expected |
| - | - | - | - | 0 | 0 | 1 | 0 | Resume Current Upgrade Session |
| - | - | - | - | 0 | 0 | 1 | 1 | Start Recovery Procedure |
| - | - | - | - | 0 | 1 | 0 | 0 | Abort Current Upgrade Session |
| - | - | - | - | 1 | 0 | 0 | 0 | Get Upgrade Session Status |
| - | x | x | - | - | - | - | - | RFU |

If P1 indicates “Start New Upgrade Session” (P1.b1=1), then:

- This command is known as a MANAGE ELF UPGRADE [start] command.
- If the upgrade session status is other than NO_UPGRADE_SESSION, then:
  - If the upgrade session status is UPGRADE_COMPLETED, then the current session shall be closed (i.e. any remaining session data shall be discarded, including any Upgrade Confirmation if available) and a new upgrade session shall be started.
  - Otherwise an error status shall be returned.
- If P1 indicates “Start ELF Upgrade Session – More ELF Upgrade Requests expected”, then:
  - If the implementation does not support upgrading more than one ELF within the same ELF Upgrade Session, then an error status shall be returned.
  - The data field of this command contains an ELF Upgrade Request, which is a request to upgrade a specific ELF during this ELF Upgrade Session, and at least one subsequent MANAGE ELF UPGRADE [start] command will follow to request the upgrade of another ELF during the same ELF Upgrade Session.
  - Notice that the new ELF Upgrade Session shall only be deemed started after the last MANAGE ELF UPGRADE [start] command has been received and all ELF upgrade requests have been recorded and accepted.
- If P1 indicates “Start ELF Upgrade Session – Last (or only) ELF Upgrade Request”, then:
  - The data field of this command contains the last ELF Upgrade Request to be processed during this ELF Upgrade Session.

If P1 indicates “Resume Current Upgrade Session”, then:

- This command is known as a MANAGE ELF UPGRADE [resume] command.

If P1 indicates “Start Recovery Procedure”, then:

- This command is known as a MANAGE ELF UPGRADE [recovery] command and is intended to forcefully start the ELF Upgrade Recovery Procedure (see section 3.2.4.2) even if the Restore Phase hasn’t been entered.

- If the upgrade session status is other than WAITING_EXECUTABLE_LOAD_FILE, then an error status shall be returned.

If P1 indicates “Abort Current Upgrade Session”, then:

- This command is known as a MANAGE ELF UPGRADE [abort] command.
- If the upgrade session status is NO_UPGRADE_SESSION or UPGRADE_COMPLETED, then an error status shall be returned.

If P1 indicates “Get Upgrade Session Status”, then:

- This command is known as a MANAGE ELF UPGRADE [status] command.

#### 4.1.2.2 Reference Control Parameter P2

The reference control parameter P2 shall be set to '00'.

#### 4.1.2.3 Data Field Sent in the Command Message

The data field of the MANAGE ELF UPGRADE command message shall be TLV encoded as described in the following table.

**Table 4-3: MANAGE ELF UPGRADE Command Data Field**

| Tag | Length | Description | Presence |
|---|---|---|---|
| 'A1' | Var. | ELF Upgrade Request | Conditional |
| '4F' | 5-16 | AID of old ELF version (i.e. the one being upgraded) | Mandatory |
| '4F' | 5-16 | AID of new ELF version (i.e. replacing the old one) | Optional |
| '80' | 1 | Upgrade Options (see Table 4-4) | Optional |
| '81' | 2 | Minimum ELF version number for the old ELF version | Optional |
| 'C7' | 2 or 4 | Volatile data Minimum Memory Requirement | Optional |
| 'C8' | 2 or 4 | Non-volatile data Minimum Memory Requirement | Optional |
| '82' | 32 | Upgrade Session Digest | Conditional |
| 'B6' | Var. | Control Reference Template for Digital Signature | Conditional |
| '42' | 1-n | Identification Number of the SD with Token Verification privilege | Optional |
| '45' | 1-n | Image number of the SD with Token Verification privilege | Optional |
| '5F20' | 1-n | Application Provider identifier | Optional |
| '93' | 1-n | Token identifier/number (digital signature counter) | Conditional |
| '9E' | 1-n | Upgrade Token | Conditional |

Tag 'A1' shall be present if P1.b1=1 (Start New Upgrade Session), otherwise it shall be absent. If tag 'A1' is present, then sub-tags shall be used as follows:

- At least one occurrence of sub-tag '4F' shall be present and used to specify the AID of the ELF that shall be upgraded.

- A second occurrence of sub-tag '4F' may be present and used to specify the AID of the new ELF version (i.e. replacing the old one) if different. If this second occurrence is not present, then the ELF Upgrade Session shall assume that the new ELF version will have the same AID as the old ELF version.
- Sub-tag '80' may be used to specify various options for the ELF Upgrade Session (see Table 4-4).
- Sub-tag '81' may be used to specify a required minimum version number for the old ELF version. This parameter may be used to ensure that the new ELF version will be able to restore instance data from the old ELF version. If the version number of the old ELF version is lower than the specified minimum version number, then an error shall be returned.
- Sub-tags 'C7' and 'C8' specify the minimum amount of volatile memory and non-volatile memory, respectively, that must be available on the card at the time this command is received. This parameter may prevent the ELF Upgrade Session from failing due to a lack of memory. If the required minimum amount of memory is not available, then an error shall be returned.

Tag '9E' shall be present if P1.b1=1 and P1.b5=0 (Start New Upgrade Session – Last (or only) ELF Upgrade Request) and the Security Domain processing this command has the Delegated Management privilege; otherwise it shall be absent. See section 4.1.2.5 for a description of the Upgrade Token.

For implementations based on v1.0, tag '82' shall not be present. Tag '82' (Upgrade Session Digest) shall be present when tag '9E' is present and more than one ELF is being upgraded. For implementations based on v1.1 (or higher) of this specification, tag '82' may be present even if a single ELF is upgraded. The Upgrade Session Digest shall be a SHA-256 digest of the list/concatenation of the ELF Upgrade Requests (i.e. occurrences of tag 'A1') recorded for this session. The entity responsible for generating an Upgrade Token (tag '9E') for this command shall check that this digest is indeed a SHA-256 digest of the list/concatenation of the ELF Upgrade Requests present in the MANAGE ELF UPGRADE [start] command(s) that it is provided with. On the card side, the implementation shall compute a SHA-256 digest of the concatenation of all the received ELF Upgrade Requests (i.e. in the first and up to the last (or only) MANAGE ELF UPGRADE [start] command) and compare it to the received Upgrade Session Digest. If the comparison fails, the upgrade session shall be aborted.

Tag 'B6' should be present when tag '9E' is present.

The “Upgrade Options” TLV shall be coded as described in the following table.

**Table 4-4: Coding of Upgrade Options TLV**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 0 | Legacy. Shall be set to 0. |
| - | - | - | - | - | - | x | - | RFU |
| - | - | - | - | - | 1 | - | - | Trigger Restore Phase automatically |
| 1 | - | - | - | - | - | - | - | Automatically resume after power loss |
| - | x | x | x | x | - | - | - | RFU |

Multiple options may be specified.

If option bit b3=1, then the Restore Phase shall start automatically as soon as the expected ELF version(s) (new ELF version(s), or old ELF version(s) if the process is in state WAITING_RESTORE_FAILED) is (are) detected on the card. If several ELF Upgrade Requests have been recorded for this ELF Upgrade Session, the Restore Phase shall only start after all expected new ELF versions have been detected on the card.

If option bit b8=1, then the ELF Upgrade Process shall try to automatically resume after a power loss and continue as far as possible (i.e. until completion or next WAITING_XXX state). If a MANAGE ELF UPGRADE command (e.g. Get Upgrade Session Status) is received before the process is resumed, then the scheduled automatic resumption shall be discarded. This option does not guarantee exactly when the process will be resumed upon card reset (e.g. resumption may be scheduled to start after a short period of time so as not to impact boot operations) or whether a specific implementation will actually resume the process at all (e.g. an implementation may choose not to impact a transaction started over the contactless interface); such details are implementation-dependent. This option is recommended unless the off-card entity intends to abort the process in case of interruption (and would like to prevent the completion of the current phase).

Finally, if several ELF Upgrade Requests are being submitted for an ELF Upgrade Session, the implementation shall ensure that the exact same options are specified for each ELF Upgrade Request (i.e. the Upgrade Options TLV shall have the same value in each request). If not, an error status shall be returned.

#### 4.1.2.4 About Library Executable Load Files

A Library ELF is an ELF that does not define any Application module and consequently for which no data needs to be saved (resp. restored). If the implementation supports upgrading multiple ELFs within the same session, the MANAGE ELF UPGRADE [start] command may specify a Library ELF to be part of the upgrade session. During the upgrade session, the processing of such a Library ELF is trivial as there is no data to be saved (resp. restored) and no Application instance to be taken care of. Adding Library ELFs to the list of upgraded ELFs is a bit more consistent from a procedural perspective and also has the theoretical advantage of eliminating the risk of residual dynamic dependencies (see section 3.2.2) being created during the Saving Phase and later discovered during the Loading Phase (i.e. as all dependencies between upgraded ELFs, including Library ELFs, will have been checked during the Saving Phase).

#### 4.1.2.5 Upgrade Token

The Upgrade Token is a signature authorizing the upgrade of one or several ELFs on the card. The OPEN shall request the Security Domain with Token Verification privilege to verify this Token. The Upgrade Token shall be computed as described in [GPCS] section C.4, with the following input data.

**Table 4-5: Input Data for Upgrade Token Computation**

| Name | Length |
|---|---|
| Reference control parameter P1 | 1 |
| Reference control parameter P2 | 1 |
| Length of the following data fields (value of the Lc field prior to a Token being added) | 1 |
| Last (or only) ELF Upgrade Request (tag 'A1' / full TLV structure) | Var. |
| Upgrade Session Digest (tag '82' / full TLV structure) | Var. |
| Control Reference Template for Digital Signature (tag 'B6' / full TLV structure) | Var. |

### 4.1.3 Response Message

#### 4.1.3.1 Data Field Returned in the Response Message

The response data field shall be coded as described in the following table.

**Table 4-6: MANAGE ELF UPGRADE Response Data Field**

| Name | Length | Value | Presence |
|---|---|---|---|
| Length of Upgrade Confirmation | 1-2 | '00' - '7F' or '81 80' - '81 FF' | Mandatory |
| Upgrade Confirmation | 0-n | 'xxxx...' – see [GPCS] section 11.1.6 | Conditional |
| Length of ELF Upgrade Session Info | 1-3 | '00' - '7F' or '81 80' - '81 FF' or '82 01 00' - '82 FF FF' | Mandatory |
| ELF Upgrade Session Info | Var. | See Table 4-7 | Mandatory |

The Upgrade Confirmation shall be present if the Security Domain processing this command has the Delegated Management privilege and the ELF Upgrade Session is in the UPGRADE_COMPLETED state; otherwise it shall be absent (i.e. length set to 0). The Upgrade Confirmation includes an Upgrade Receipt that shall be computed as described in section 4.1.3.2.

**Table 4-7: Structure of ELF Upgrade Session Info**

| Tag | Length | Description | Presence |
|---|---|---|---|
| 'A1' | Var. | ELF Upgrade Information | Mandatory |
| '90' | 1 | ELF Upgrade Session Status (see Table 4-8) | Mandatory |
| '4F' | 5-16 | AID of old ELF version (i.e. the one being upgraded) | Conditional |
| '4F' | 5-16 | AID of new ELF version (i.e. replacing the old one) | Conditional |
| '80' | 1 | Upgrade Options (see Table 4-4) | Conditional |
| 'A1' | Var. | ELF Upgrade Information | Conditional |
| … | … | … | … |

If P1 indicates “Get Upgrade Session Status” and the ELF Upgrade Session Status is other than NO_UPGRADE_SESSION, then:

- The ELF Upgrade Session Info shall include one ELF Upgrade Information (tag 'A1') for each ELF Upgrade Request processed in the ELF Upgrade Session.
- For every occurrence of ELF Upgrade Information:
  - Tags '4F' (possibly two occurrences) shall be present.
  - The ELF Upgrade Session Status (tag '90') shall be present and have the same value.
  - The Upgrade Options (tag '80') shall be present and have the same value.
- If the response exceeds 256 bytes, it shall be chained as described in section 11.1.5.2 of [GPCS] using the '61xx' status word, and multiple GET RESPONSE commands should be used to retrieve the remaining response data.

Otherwise the ELF Upgrade Session Info shall include only a single occurrence of ELF Upgrade Information (tag 'A1') itself including only the ELF Upgrade Session Status (tag '90').

The coding of the ELF Upgrade Session Status is described in the following table.

**Table 4-8: Coding of ELF Upgrade Session Status**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | NO_UPGRADE_SESSION |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | UPGRADE_COMPLETED |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | WAITING_EXECUTABLE_LOAD_FILE |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | WAITING_RESTORE |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | WAITING_RESTORE_FAILED |
| - | - | - | - | x | - | - | - | RFU |
| x | x | x | x | 0 | 0 | 0 | 0 | INTERRUPTED |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | INTERRUPTED_SAVING |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | INTERRUPTED_CLEANUP |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | INTERRUPTED_DELETE |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | INTERRUPTED_INSTALL |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | INTERRUPTED_RESTORE |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | INTERRUPTED_CONSOLIDATE |

See section 3.3 for a description of the status values.

If the ELF Upgrade Session is in the UPGRADE_COMPLETED state, then the session status shall automatically transition to NO_UPGRADE_SESSION after the response to this command has been sent (NOTE: the response itself shall indicate the UPGRADE_COMPLETED state). Notice therefore that the indication of the upgrade session being completed may be retrieved only once. If the off-card entity could not get such an information (e.g. communication failure), it may still confirm that the upgrade succeeded or failed using a GET STATUS command and inspecting packages and their version numbers.

#### 4.1.3.2 Upgrade Receipt

The Upgrade Receipt is a signature providing confirmation that a successful ELF upgrade occurred through the delegated ELF Upgrade Process. The Upgrade Receipt shall be computed as described in [GPCS] section C.5, with the input data shown in the following table

**Table 4-9: Input Data for Upgrade Receipt Computation**

| Name | Length |
|---|---|
| Confirmation Data as defined in [GPCS] section 11.1.6 | 1-n |
| Length of Upgrade Session Digest | 1 |
| Upgrade Session Digest | 32 |

The value of the Upgrade Session Digest shall be the one read from tag '82' found in the last (or only) MANAGE ELF UPGRADE [start] command that started the upgrade session.

#### 4.1.3.3 Processing State Returned in the Response Message

A successful execution of the command shall be indicated by status bytes '90 00'.

This command may return either a general error condition as listed in [GPCS] section 11.1.3 or one of the following error conditions.

**Table 4-10: MANAGE ELF UPGRADE Error Conditions**

| SW1 | SW2 | Meaning |
|---|---|---|
| '64' | '00' | Unknown execution error – Upgrade process aborted |
| '64' | '01' | ELF version number lower than specified minimum – Upgrade process aborted |
| '64' | '02' | Incorrect Security Domain for new ELF version – Upgrade process aborted |
| '64' | '03' | Missing modules in new ELF version – Upgrade process aborted |
| '64' | '10' | Attempt to save custom objects – Upgrade process aborted |
| '69' | '00' | Cannot start ELF Upgrade Session – Upgrade session already ongoing |
| '62' | '00' | (Warning) Unknown execution error – Upgrade Recovery Procedure started |
| '62' | '01' | (Warning) Upgrade process completed using Recovery Procedure |
| '62' | '02' | (Warning) Incorrect Security Domain for new ELF version – Upgrade Recovery Procedure started |
| '62' | '03' | (Warning) Missing modules in new ELF version – Upgrade Recovery Procedure started |

# Annex A Java Card Programming Interface

## A.1 Overview

A new `org.globalplatform.upgrade` package (v1.0), the Upgrade API, defines classes and interfaces supporting the ELF Upgrade Process.

### A.1.1 The `UpgradeManager` Class

The `UpgradeManager` class defines the following static methods:

- The `isUpgrading()` method allows an Application instance to check whether it is invoked during the ELF Upgrade Process. This method returns true only for Application instances involved in an ongoing ELF Upgrade Session, false otherwise. It shall be used in the `Applet.install()` method to check whether an Application instance is being installed or upgraded.
- The `getPreviousPackageVersion()` method allows an Application instance to check the version (major + minor) of the previous package version (i.e. the one being upgraded). This method may help manage migration from different package versions.
- The `checkPreviousPackageAID(byte[], short, byte)` method allows an Application instance to check a specified AID against the AID of the previous package version (i.e. the one being upgraded). This method may help manage migration from different package versions.
- The `createElement(byte, short, short)` method allows the creation of `Element` instances. See section A.1.3. An `Element` instance created by this method belongs to the caller Application instance.
- The `matchMappedElement(Object o)` method allows looking for a `MappedElement` instance mapped to the specified custom object. See section A.2.3.

### A.1.2 The `OnUpgradeListener` Interface

The `OnUpgradeListener` interface shall be implemented by Applet subclasses that wish to perform specific save or restore operations. This interface does not extend the Shareable interface, as it will be invoked by the system directly during the upgrade process.

- `public Element onSave();`

  This method allows the Application instance to save its instance data in a hierarchy of `Element` instances and to return its root `Element` instance that shall be saved by the system. See section A.1.3.

  This method shall avoid modifying objects (e.g. clearing key objects or resetting objects) so that objects may be correctly migrated, or the Application instance may correctly return to its initial state in case of abort during the Data Saving Sequence.

  In case of interruption (e.g. power loss) during the execution of this method, the Java Card Garbage Collector will be executed automatically to collect lost `Element` instances.

- `public void onCleanup();`

  This method allows the Application instance to perform some cleanup actions that might be required to facilitate its deletion (e.g. releasing dynamic dependencies preventing the deletion).

  This method shall avoid performing unnecessary modifications (as discussed for the `onSave()` method).

- `public void onRestore(Element root);`

  This method allows the Application instance to restore its instance data from a root `Element` provided by the system. See section A.1.3.

### A.1.3 The `Element` Interface

A custom object is an instance of a custom class, i.e. a class belonging to any of the packages currently being upgraded. References to custom objects cannot be saved directly because the class hierarchies of the new package versions may be quite different, so that custom objects might no longer make sense in the new package versions (i.e. it would not be possible to restore objects created from classes that no longer exist). Therefore, it is necessary to save their content (i.e. fields) differently.

The `Element` interface is introduced for that purpose. An `Element` instance represents a generic object (with primitive fields and object fields) that shall be used to save the fields of a custom object. In other words, a custom object that you would like to save but cannot save shall be mapped to an `Element` instance that can be saved.

```
public interface Element {

   public Element write(boolean z); // size:1
   public Element write(byte b); // size:1
   public Element write(short s); // size:2
   public short canWriteBoolean();// number of booleans that may be written
   public short canWriteByte();// number of bytes that may be written
   public short canWriteShort();// number of shorts that may be written

   public Element write(Object o);
   public short canWriteObject();// number of objects that may be written

   public void initRead(); // reset read pointers (primitive data + objects)

   public boolean readBoolean();
   public byte readByte();
   public short readShort();
   public short canReadBoolean();// number of booleans that may be read
   public short canReadByte();// number of bytes that may be read
   public short canReadShort();// number of shorts that may be read

   public Object readObject();
   public short canReadObject();// number of objects that may be read

}
```

A typical example of a custom object is the Application instance itself (which is requested to save its data) that shall be mapped to an `Element` instance. The `Element` instance will be the root of the saved instance data hierarchy for that Application instance. In the best case, you may only have to create the single `Element` instance corresponding to your Application instance.

An `Element` instance is created using the `UpgradeManager.createElement(…)` method. To do so, you shall first compute the "dimensions" of the custom object whose contents you want to save, i.e. the size required to save primitive data (of type `boolean, byte, short`) and the number of object references that need to be saved. The different `Element.write(…)` methods then allow filling your `Element` instance with the instance data of your custom object:

- Using the `Element.write(Object)` method, it is possible to save any object reference except a reference to a custom object. It is possible to save the null reference also. For example, it is possible to save references to arrays of primitive data, arrays of objects (but not arrays of custom objects), and objects created from classes of the standard Java Card API implementation (which is enough to cover all situations). It is important to save such non-custom objects directly whenever possible and avoid creating `Element` instances unnecessarily in order to reduce memory usage during the ELF Upgrade Process. Attempts to save custom objects during the Saving Phase shall be detected no later than the end of the Cleanup Sequence and shall result in the automatic abortion of the ELF Upgrade Process. In this case, a specific status word shall be returned to clearly indicate the reason for the abortion.
- For security reasons, the implementation shall separate primitive data and object references when saved and restored, so that an object reference may not be forged from primitive data or, conversely, may not be disclosed as primitive data when reading the contents of the `Element` instance during the Restore Phase. In particular, during the Restore Phase, the `readObject()` method will only read from stored objects and other `readXXX()` methods will only read from stored primitive data, as shown in the example below:

```
write(short1);
write(object1);
write(short 2);
write(object2);
...
readObject(); // read object1
readShort();  // read short1
readObject(); // read object2
readShort();  // read short2
```

- There is no such storage separation defined for data of different primitive types. Therefore, a particular implementation of the Upgrade API may allow writing of some primitive data and later reading it assuming a different primitive type (e.g. write a `byte` value and later read it as a `boolean` value, or write a `short` value and later read it as two distinct `byte` values). However, supporting such behavior and the results of such conversions remain out of scope of this specification and it is recommended to read primitive data without attempting such type conversions.
- Saving object references may create residual dynamic dependencies. See examples in section 3.2.2.

Finally, during the Restore Phase, the `Element.initRead()` method must be invoked once on each `Element` instance to reset read pointers before any call to a `readXXX()` method is performed. Indeed, if the Restore Phase is resumed after a power loss, an Application instance may fall upon an `Element` instance from which data has already been read, so read pointers shall be reset in order to read its data again correctly. Notice that we don't have to deal with the same problem during the Saving Phase as, in the case of power loss, previously created `Element` instances will be lost (and garbage collected) and new `Element` instances will be created.

## A.2 Using the `Element` Interface

Let's assume we defined custom classes A and B as follows:

```
class A {
  boolean z; // primitive value
  S s; // standard library class or class from other package
  byte[] bArray;
  B b; // custom class B
  A a1;
  A a2;
}

class B {
  byte v1;
  short v2;
}
```

Instances of class A may be organized in a hierarchy (using the `a1` and `a2` fields).

### A.2.1 During the Saving Phase

Class B implements static method `B.save(B)` returning an `Element` instance:

```
static Element save(B b) {
  if (b == null)
    return null;
  return UpgradeManager.createElement(Element.TYPE_SIMPLE,3,0)
    .write(b.v1).write(b.v2);
}
```

Class A implements static method `A.save(A)` returning an `Element` instance:

```
static Element save(A a) {
  if (a == null)
    return null;
  return UpgradeManager.createElement(Element.TYPE_SIMPLE,1,5)
    .write(a.z) // primitive value
    .write(a.s) // standard library class or class from other package
    .write(a.bArray)
    .write(B.save(a.b)) // custom class B
    .write(A.save(a.a1))
    .write(A.save(a.a2));
}
```

Notice how recursion is stopped when a `null` field is saved (first two lines).

If you have a deep hierarchy of objects, e.g. 10 levels, you will need your save method to recurse deeply, e.g. 10 times. In order to maximize the possible depth of recursion, your save method should minimize its number of local variables, as in the above example.

Using this pattern, developers can simply write a method to map their Application instance to an `Element` instance that will be saved:

```
public Element onSave() {
  return MyApplet.save(this);
}
```

In order to reduce memory usage during the ELF Upgrade Process, you will sometimes want to create as few `Element` instances as needed. First, remember that only custom objects need to be mapped to `Element` instances. Then some situations may be treated in different ways. For example, as the B class is very simple, you may consider that it is not worth creating an `Element` instance for a B object and may instead save its fields directly:

```
static Element save(A a) {
  if (a == null)
    return null;
  Element e = UpgradeManager.createElement(Element.TYPE_SIMPLE,1,5)
    .write(a.z) // primitive value
    .write(a.s) // standard library class or class from other package
    .write(a.bArray);
  if (a.b != null)
    e.write(a.b.v1).write(a.b.v2);
  return e.write(A.save(a.a1)).write(A.save(a.a2));
}
```

### A.2.2 During the Restore Phase

In the new package version, class A is upgraded to class C and class B is upgraded to class D.

Class D implements static method `D.restore(Element)` returning a D instance:

```
static D restore(Element e) {
  if (e == null)
    return null;
  D d = new D();
  e.initRead();
  d.f1 = e.readByte();
  d.f2 = e.readShort();
  return d;
}
```

Class C implements static method `C.restore(Element)` returning a C instance:

```
static C restore(Element e) {
  if (e == null)
    return null;
  C c = new C();
  e.initRead(); // in case of power loss
  c.f1 = e.readBoolean();
  c.f2 = e.readObject(); //s (directly)
  c.f3 = e.readObject(); //bArray (directly)
  c.f4 = D.restore(e.readObject()); //b (mapped to Element)
  c.f5 = C.restore(e.readObject()); //a1 (mapped to Element)
  c.f6 = C.restore(e.readObject()); //a2 (mapped to Element)
  return c;
}
```

Notice how recursion is stopped when a `null` field is restored (two first lines).

If you have a deep hierarchy of objects, e.g. 10 levels, you will need your restore method to recurse deeply, e.g. 10 times. In order to maximize the possible depth of recursion, your restore method should minimize its number of local variables, as in the above example.

Using this pattern, developers can simply write a method to restore the data of a previous Application instance from an `Element` provided by the system:

```
public void onRestore(Element root) {
  this.restore(root); // this == myAppletInstance
}
```

The Application instance shall not keep any reference to any `Element` instance it may manipulate during the Restore Phase, as `Element` instances will become useless after the end of the upgrade process, and this would block memory space unnecessarily.

### A.2.3 Managing Duplicates and Cycles

During the Saving Phase, Application Developers should/shall detect duplicate references. Typically, when stumbling upon the same custom object reference (twice or more), you wouldn't want to create a distinct `Element` instance for it, but would rather want to reference the `Element` instance you already created. In other words, you want to enforce a unique `Element` instance for each saved object. In the same manner, during the Restore Phase, you will want to detect when an `Element` instance has already been processed to create a new custom object.

Cycles are a special case of duplicates. For example, if you have `[a -> b -> c -> a]` and you start saving `a`, then when saving `c` you will end up trying to save `a` again.

Fortunately, duplicates and cycles might not be common in your application (especially talking about smart card development). Or the Application Developer might know how to manage them easily. For example, a simple case of cycle is the cross-referencing of objects, i.e. `[a -> b -> a]` which may be used for performance reasons. In this case, the Application Developer may easily manage saving `[a -> b]` only and may restore both `[a -> b]` and `[b -> a]` during the Restore Phase.

Nevertheless, if you really have to manage duplicates and cycles, which your custom class A may allow by design, you may use the additional helper methods of the `MappedElement` interface:

```
public interface MappedElement extends Element {
  public Object getMappedObject; //helper (restore)
  public Element setMappedObject(Object o); //helper (save/restore)
}
```

The coding patterns are then as follows:

**= SAVING PHASE ===**

```
static Element save(A a) {
  if (a == null)
    return null;
  Element e = UpgradeManager.matchMappedObject(a);
  if (e != null)
    return e;
  return ((MappedElement)UpgradeManager.createElement(Element.TYPE_MAPPED, 1,
5))
      .setMappedObject(a)
      .write(a.z)
      .write(a.s)
      .write(a.bArray)
      .write(B.save(a.b))
      .write(A.save(a.a1))
      .write(A.save(a.a2));
}
```

The `matchMappedObject()` static method iterates over all `Element` instances to check whether any of these instances was previously associated with object `a`.

Notice that the call to `setMappedObject(a)` is the first thing done after the creation of the new `Element` instance, as following calls may fall upon `a` again.

All mapped object references (stored with `setMappedObject()`) are reset at the end of the Saving Phase (the referenced objects will be deleted).

**= RESTORE PHASE ===**

```
static C restore(Element e) {
  if (e == null)
    return null;
  if (e.getMappedObject() != null)
    return e.getMappedObject();
  C c = new C();
  e.setMappedObject(c);
  e.initRead();
  c.f1 = e.readBoolean();
  c.f2 = e.readObject(); //s
  c.f3 = e.readObject(); //bArray
  c.f4 = D.restore(e.readObject()); //b
  c.f5 = C.restore(e.readObject()); //a1
  c.f6 = C.restore(e.readObject()); //a2
  return c;
}
```

Notice that the call to `setMappedObject(c)` is the first thing done after the creation of the new `C` instance, as following calls may fall upon `e` again.

### A.2.4 Saving a Field of Interface Type

If you need to save a field of an interface type that may store an object created from different implementation classes, then save (e.g. as a byte) the actual type of the object before saving the object itself.

### A.2.5 Design Guidelines for Minimizing Required Upgrade Memory Space

If possible, try to store your primitive data in a primitive array (and access it with getter/setter methods). Doing so, your primitive array may be saved as a single object and you won't need your `Element` instance to allocate an additional primitive array to store (copy) all your primitive fields. Of course, this kind of design may impact the performance of your application.

For the same reason, you could try to store all your non-custom objects in a single object array (and access them with getter/setter methods) so that you may save all of them in a single operation. However, you will still have to process custom objects separately. Try not to mix custom and non-custom objects in the same array.

Transient arrays don't need to be saved. Instead, you should simply save the length of your transient array, and if needed create a transient array of the same length during the Restore Phase.

If you have to deal with a significant number of custom objects and you care about the number of `Element` instances that may be required to save their contents, then consider using the Single Element Instance pattern (see section A.2.6), which uses a single `Element` instance for your entire Application instance.

### A.2.6 The Single Element Instance Pattern

If you really care about minimizing the amount of memory space required for working data (i.e. `Element` instances) during the upgrade process, you may wish to use the Single Element Instance pattern where only a single root `Element` instance is needed per Application instance. Notice however that this pattern does not provide support for managing duplicates and cycles.

**= SAVING PHASE ===**

This pattern creates a unique `Element` instance in the `onSave()` method of the Application instance, i.e. the root `Element` instance. This pattern requires specific methods to compute the total size of primitive data and total count of object references that shall be stored by the root `Element` instance.

```
public Element onSave() {
  Element root = UpgradeManager.createElement(Element.TYPE_SIMPLE,
    MyApplet.computePrimitiveDataSize(this),
    MyApplet.computeObjectCount(this));
  MyApplet.save(root,this);
  return root;
}
```

Each custom class should implement static methods to compute the size of primitive data and the count of object references for any of its instances. Of course, a size/count of 0 shall be returned for a null reference.

```
class B {
  ...
  static short computePrimitiveDataSize(B b) {
    if (b == null)
      return 0;
    return (short)(Element.SIZE_BYTE+Element.SIZE_SHORT); // 3
  }

  //static short computeObjectCount(B b) { // known to be 0
  //   return 0;
  //}
  ...
}

class A {
  ...
  static short computePrimitiveDataSize(A a) {
    if (a == null)
      return 0;
    return Element.SIZE_BOOLEAN // 1
      + B.computePrimitiveDataSize(a.b)
      + A.computePrimitiveDataSize(a.a1)
      + A.computePrimitiveDataSize(a.a2);
  }

  static short computeObjectCount(A a) {
    if (a == null)
      return 0;
    return 5 // number of object fields in class A
      //+ B.computeObjectCount(a.b) // known to be 0
      + A.computeObjectCount(a.a1)
      + A.computeObjectCount(a.a2);
  }
  ...
}
```

Static save methods implemented by custom classes shall no longer create `Element` instances. Instead, the root `Element` instance shall be passed as an argument to each of these static save methods.[1]

```
class B {
  ...
  static void save(Element e, B b) {
    if (b == null) {
      e.write(null);
      return;
    }
    e.write(UpgradeManager.NonNullReference); // (B)
    e.write(b.v1).write(b.v2);
  }
  ...
}

class A {
  ...
  static void save(Element e, A a) {
    if (a == null) {
      e.write(null);
      return;
    }
    e.write(UpgradeManager.NonNullReference); // (A)
    e.write(a.z);
    e.write(a.s);
    e.write(a.bArray);
    B.save(e,a.b);
    A.save(e,a.a1);
    A.save(e,a.a2);
  }
  ...
}
```

As usual, null references (i.e. corresponding to empty reference fields) shall be saved. Instead of creating and saving a new `Element` instance for a specific custom object, an anonymous non-null reference (`UpgradeManager.NonNullReference`) shall be saved in the root `Element` instance as a hint (to be used during the Restore Phase) that subsequently saved data (primitive data and/or references) relates to the content of that specific custom object.

**= RESTORE PHASE ===**

During the Restore Phase, the root `Element` instance shall be passed as an argument to each of the static restore methods.[2]

```
public void onRestore(Element root) {
  root.initRead();//once only
  MyApplet.restore(root,this);
}
```

[1] Alternatively, the root `Element` instance may be temporarily stored in a field (in order to minimize the stack size in case of deep recursion). Of course, the content of that field shall not be saved.

[2] Alternatively, the root `Element` instance may be temporarily stored in a field (in order to minimize the stack size in case of deep recursion). The content of that field shall be released at the end of the `onRestore(Element)` method.

If a static restore method wishes to learn about a custom object field, it shall try to read an object. If it reads a null reference, then the custom object field was empty. If it reads a non-null reference, then this reference is the anonymous non-null reference (`UpgradeManager.NonNullReference`) saved during the Saving Phase as a hint that subsequent data (primitive data and/or references) relates to the content of a custom object stored in that specific custom object field. How much data actually relates to that custom object is a matter of convention between the Saving Phase and Restore Phase.

```
static D restore(Element e) {
  if (e.readObject() == null)
    return null;
  // UpgradeManager.NonNullReference was read (B)
  D d = new D();
  d.f1 = e.readByte();
  d.f2 = e.readShort();
  return d;
}

static C restore(Element e) {
  if (e.readObject() == null)
    return null;
  // UpgradeManager.NonNullReference was read (A)
  C c = new C();
  c.f1 = e.readBoolean(); //z
  c.f2 = e.readObject(); //s
  c.f3 = e.readObject(); //bArray
  c.f4 = D.restore(e); //b
  c.f5 = C.restore(e); //a1
  c.f6 = C.restore(e); //a2
  return c;
}
```

### A.2.7 Saving an Array of Custom Objects

This pattern may be used to map an array of custom objects to a single `Element` instance (instead of saving an array of `Element` instances corresponding to that array). In short, the contents of the B objects will be saved in sequence in the same `Element` instance. The length of the array must also be saved.

In this example, a custom class T holds an array of custom objects of class B.

**= SAVING PHASE ===**

```
class T {
  ...
  B[] biz;
  ...
  static short computePrimitiveDataSize(T t) {
    if (t == null)
      return 0;
    short size = 0;
    ...
    if (t.biz != null) {
      short length = (short)t.biz.length;
      size += Element.SIZE_SHORT; // (space for array length)
      for (short i = 0; i < length; i++) {
        size += B.computePrimitiveDataSize(t.biz[i]);
      }
    }
```

```
      ...
      return size;
    }

    static void save(Element e, T t) {
      if (t == null) {
        e.write(null);
        return;
      }
      ...
      if (t.biz != null) {
        e.write(UpgradeManager.NonNullReference); // t.biz
        short length = (short) t.biz.length;
        e.write(length); // save array length
        for (short i = 0; i < length; i++) {
          if (t.biz[i] != null) {
            // non empty slot
            e.write(UpgradeManager.NonNullReference); // (B)
            e.write(t.biz[i].v1).write(t.biz[i].v2);
          }
        }
      }
      ...
    }
    ...
  }
```

**= RESTORE PHASE ===**

```
  class U {
    ...
    static U restore(Element e) {
      if (e.readObject() == null)
        return null;
      // UpgradeManager.NonNullReference was read (T)
      U u = new U();
      ...
      if (e.readObject() != null) {
        // restore t.biz
        short length = e.readShort(); // read array length
        u.biz = new D[length];
        for (short i = 0; i < length; i++) {
          if (e.readObject() != null) {
            // non empty array slot
            D d = new D();
            d.f1 = e.readByte(); // b.v1
            d.f2 = e.readShort(); // b.v2
            u.biz[i] = d;
          }
        }
      }
      ...
      return u;
    }
    ...
  }
```

# Annex B ELF Upgrade Sample Scenarios

## B.1 Authorized Management

In this example, the Security Domain has the Authorized Management privilege.

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start] // Start Upgrade Session
  - Saving Phase completes
  - Session State = WAITING_EXECUTABLE_LOAD_FILE
- INSTALL [for load] (New ELF version)
- LOAD (first) … LOAD (last)
  - New ELF version is found on card
  - Session State = WAITING_RESTORE
  - If automatic triggering of Restore Phase was selected:
    - Restore Phase starts automatically
    - Restore Phase completes
      - Discard all session data
      - Session State = UPGRADE_COMPLETED
- MANAGE ELF UPGRADE [resume] // Finalize Upgrade Session
  - If automatic triggering of Restore Phase was **not** selected:
    - Restore Phase starts
    - Restore Phase completes
      - Discard all session data
      - Session State = UPGRADE_COMPLETED
  - Session State = NO_UPGRADE_SESSION

## B.2 Delegated Management

In this example, the Security Domain has the Delegated Management privilege.

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start] // Start Upgrade Session
  - **Upgrade Token is verified**
  - Saving Phase completes
  - Session State = WAITING_EXECUTABLE_LOAD_FILE
  - **No confirmation is returned**
- INSTALL [for load] (New ELF version)
  - **Load Token is verified**
- LOAD (first) … LOAD (last)
  - New ELF version is found on card
  - Session State = WAITING_RESTORE
  - **Load Confirmation is returned**
  - If automatic triggering of Restore Phase was selected:
    - Restore Phase starts automatically
    - Restore Phase completes
      - Discard all session data **(except data related to the Upgrade Confirmation)**
      - Session State = UPGRADE_COMPLETED
- MANAGE ELF UPGRADE [resume] // Finalize Upgrade Session
  - If automatic triggering of Restore Phase was **not** selected:
    - Restore Phase starts
    - Restore Phase completes
      - Discard all session data
      - Session State = UPGRADE_COMPLETED
  - Session State = NO_UPGRADE_SESSION
  - **Upgrade Confirmation is returned**

## B.3 Failure during Restore, Reload Old ELF Version

In this example, a fatal error occurs using the new ELF version during the Restore Phase and the ELF Upgrade Process enters the ELF Upgrade Recovery Procedure. The ELF Upgrade Process then waits for the old ELF version to be reloaded. As soon as the old ELF version is detected on the card, the ELF Upgrade Process can proceed with restoring the initial state.

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start] // Start Upgrade Session
- INSTALL [for load] (New ELF version)
- LOAD (first) … LOAD (last)
- MANAGE ELF UPGRADE [resume] // Finalize Upgrade Session
  - Restore Phase starts
  - **Restore Phase fails**
    - **New ELF version and all new Application instances are deleted**
    - **Session State = WAITING_RESTORE_FAILED**
- **INSTALL [for load] (Old ELF version)**
- **LOAD (first) … LOAD (last)**
  - **Old ELF version is found on card**
  - **Session State = WAITING_RESTORE**

… (Restoration is performed using old ELF version) …

⇨ **Initial state prior to ELF Upgrade Session has been restored**

## B.4 Void

## B.5 Load New ELF Version before Upgrade Session

In this example, the off-card entity loads the new ELF version in advance so that the upgrade session may be able to complete automatically with a single command. This may only be done if the old and new ELF versions do not share the same AID. It is assumed that the card has enough memory to store the new and old ELF versions at the same time.

- Open a Secure Channel session
- **INSTALL [for load] (New ELF version)**
- **LOAD (first) … LOAD (last)**
- MANAGE ELF UPGRADE [start] // Start Upgrade Session
  - Saving Phase starts
  - Saving Phase completes
  - **New ELF version is found on card**
  - **Session State = WAITING_RESTORE**
  - If automatic triggering of Restore Phase was selected:
    - Restore Phase starts automatically
    - Restore Phase completes
      - Discard all session data
      - Session State = UPGRADE_COMPLETED
- …

Notice that in this scenario, if automatic triggering of Restore Phase is selected and no error occurs, then the entire upgrade session can fit in a single APDU command (MANAGE ELF UPGRADE [start]).

## B.6 New ELF Version Requires Library Update

In this example, the old ELF version imports a Library ELF that is also used by the new ELF version and the Library ELF needs to be updated before loading the new ELF version.

If the implementation doesn't support upgrading multiple ELFs within the same upgrade session, the Library ELF may only be replaced (i.e. deleted then reloaded) during the Loading Phase:

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start] // Start Upgrade Session
  - Saving Phase starts
    - **Objects created from Library classes shall not be saved directly, otherwise it will not be possible to delete the Library ELF in the coming steps.**
  - Saving Phase completes
  - Session State = WAITING_EXECUTABLE_LOAD_FILE
- **DELETE (old Library version)**
- **INSTALL [for load] (new Library ELF version)**
- **LOAD (first) … LOAD (last) // new Library ELF version is loaded**
- INSTALL [for load] (new ELF version)
- LOAD (first) … LOAD (last)
  - New ELF version is found on the card
  - Session State = WAITING_RESTORE
- …

If the implementation supports upgrading multiple ELFs within the same upgrade session, it is possible to specify the Library ELF as part of the ELFs being upgraded and the scenario becomes simpler:

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start] (first) // ELF
- **MANAGE ELF UPGRADE [start] (last) // Library ELF**
  - Saving Phase completes
  - Session State = WAITING_EXECUTABLE_LOAD_FILE
- **INSTALL [for load] (New Library ELF version)**
- **LOAD (first) … LOAD (last) // new Library ELF version is loaded**
- INSTALL [for load] (New ELF version)
- LOAD (first) … LOAD (last)
  - All new ELF versions found on the card
  - Session State = WAITING_RESTORE
- …

# Annex C Remote Administration (Informative)

This annex describes a number of strategies that may be used by Remote Administration Servers (RAS) to solve issues they may encounter during an ELF Upgrade Session.

## C.1 Upgrade Options

In this annex, only a number of Upgrade Options are retained (see Table 4-4 for details) to limit the number of possible scenarios and to describe a limited number of strategies that the RAS may implement and that should always work. Therefore:

- Option “Trigger Restore Phase automatically” may be used but requires considering command flows that are not described in this annex and doesn't bring any benefit in the context of remote administration, as remote scripts may contain the necessary “resume” commands that should be successfully executed if the upgrade process encounters no issues.
- Option “Automatically resume after power loss” should be used to help reduce service interruption delays in case of device or card power loss. As explained in section 4.1.2.3, however, this option does not guarantee when process resumption will happen after power up, nor that it will happen at all. Using this option has no impact on the recommendations given in the following sections.

## C.2 Nominal Scenario

Given the assumptions described in section C.1, the nominal command flow used by the RAS to perform an upgrade session is as follows:

- Open a Secure Channel session
- MANAGE ELF UPGRADE [start]
  - Saving Phase completes
  - Session State = WAITING_EXECUTABLE_LOAD_FILE // Loading Phase
- INSTALL [for load] (New ELF version)
- LOAD (first) … LOAD (last)
  - New ELF version is found on card
  - Session State = WAITING_RESTORE
- MANAGE ELF UPGRADE [resume]
  - Restore Phase starts
  - Restore Phase completes
    - Discard all session data
    - Session State = UPGRADE_COMPLETED
  - Session State = NO_UPGRADE_SESSION

The RAS may wish to issue a final GET STATUS command to audit card contents after completion of the upgrade session.

Note 1: If multiple ELFs are upgraded, the command flow essentially remains the same, except that several MANAGE ELF UPGRADE [start] commands are used (each one containing an ELF Upgrade Request) and several new ELF versions are loaded during the Loading Phase.

Note 2: In a Delegated Management scenario, the command flow remains the same. The last (or only) MANAGE ELF UPGRADE [start] command would include an Upgrade Token, each INSTALL [for load] command would include a Load Token and any DELETE command would include a Delete Token. Notice that the MANAGE ELF UPGRADE [resume] command doesn't include a Token.

Note 3: An acceptable variant of this scenario consists of loading new ELF version(s) in advance before starting the ELF Upgrade Session (so they won't have to be loaded during the Loading Phase). Such a scenario may help reduce the duration of the upgrade session itself and therefore reduce service interruption delays. However, using this variant is only possible if:

- None of the new ELF version(s) shares the same AID value as any of the old ELF version(s), and
- The targeted card has enough memory space to store both new and old ELF version(s). In case of multiple ELF upgrade, the more ELFs need be upgraded, the more this constraint becomes difficult to satisfy as all the new ELF version(s) need to be loaded and fit on the card beforehand.

For these reasons, we do not expand on such a scenario in this annex.

## C.3 Memory Requirements

In the case that on-card memory space could be a potential issue for the targeted card, it is recommended that the RAS) check the minimum memory requirements by including tags 'C7' and 'C8' in the MANAGE ELF UPGRADE [start] command. Unfortunately, due to the versatility of card implementations, using these tags won't provide any solid guarantee that enough memory space is actually available on the targeted card; however, this may limit the risk and number of failures. If the requested memory sizes are not available on the targeted card, then the upgrade session will not start, meaning the RAS should use another strategy to upgrade card contents. Notice that when computing such memory requirements, given the assumptions described in section C.1, it should be taken into account that the old and new ELF version(s) won't be present at the same time on the card.

## C.4 No Response from the Card

After the start of an upgrade session, the RAS may consider that no response can be received from the card after a certain amount of time (i.e. timeout) or communication breakdown (e.g. TCP connection closed, lost SMS, card or device power loss, etc.). In such cases, the RAS doesn't know whether the command script (sent through SMS, BIP/CAT_TP, or HTTPS) was received and/or processed by the card (fully or partially), and doesn't know the status of the upgrade session. It is therefore recommended that the RAS send a new command script (whenever possible) with the following commands:

- MANAGE ELF UPGRADE [resume]
- Either a single GET STATUS command requesting information about all the packages visible to the RAS (i.e. visible on-card to the Security Domain processing the command) or more selectively, one or more GET STATUS commands requesting information about packages the RAS is interested in (i.e. old ELF version(s) and new ELF version(s)). Each GET STATUS command shall include a tag list (tag '5C') requesting that the response include tags '4F' (AID) and 'CE' (version number) for each package.

Notice that use of the Expanded Remote command structure (see [TS 102 226]) is recommended in order to receive a response for each of the APDU commands included in the command script. The Compact Remote command structure may be used but will require the RAS to send several Command scripts.

If the ELF Upgrade Session was interrupted somewhere in the Saving Phase, then the MANAGE ELF UPGRADE [resume] will resume the upgrade process until an error is detected or it reaches the Loading Phase. If interrupted in the Restore Phase, it will resume the upgrade process until an error is detected or it reaches completion. The case where the MANAGE ELF UPGRADE [resume] would return an error is described in section C.5. If the command returns with no error and the returned ELF Upgrade Session Status is:

- UPGRADE_COMPLETED or NO_UPGRADE_SESSION, then the RAS shall further inspect the response to the subsequent GET STATUS command(s) to discover:
  - Whether the upgrade session successfully completed (new ELF version(s) present with correct version number(s)), or
  - Whether the upgrade session just didn't start (old ELF version(s) present, with version number(s) other than the expected new version number(s) if old and new ELF versions share the same AID(s)).

  Notice that the UPGRADE_COMPLETED state, if returned, may pertain to another ELF Upgrade Session, so the above checks are still needed.
- WAITING_EXECUTABLE_LOAD_FILE, then the RAS shall further inspect the response to the subsequent GET STATUS command(s) to determine which new ELF version(s) still need to be loaded. The RAS should then send one or several command scripts to load the missing ELF version(s) and issue another MANAGE ELF UPGRADE [resume] command to enter the Restore Phase and complete the upgrade session.
- WAITING_RESTORE_FAILED, then the upgrade session has entered the recovery procedure and the RAS shall further inspect the response to the subsequent GET STATUS command(s) to know which old ELF version(s) still need to be reloaded. The RAS should then send one or several command scripts to load the missing ELF version(s) and issue another MANAGE ELF UPGRADE [resume] command to enter the Restore Phase and complete the recovery procedure.

See also section C.6, "About the ELF Upgrade Session's Recovery Procedure".

## C.5 Error or Warning Status Word Returned by the Card

This section recommends different strategies depending on the error/warning returned by a command script performing an upgrade session.

If a MANAGE ELF UPGRADE [start] command returns an error status word, no upgrade session was started and the RAS should further study the reasons for such a failure, retry later, or choose another strategy.

If a MANAGE ELF UPGRADE [resume] command returns a status word of '6200', '6202', or '6203', then the upgrade session entered the Restore Phase, failed, and entered the recovery procedure. The RAS should send one or more command scripts to load the old ELF version(s) and issue another MANAGE ELF UPGRADE [resume] command to enter the Restore Phase (again) and complete the recovery procedure. Subsequently, the correct completion of the recovery procedure should be indicated with a warning status word of '6201', whereas a failure of the recovery procedure should return an error status word of '6400'.

If an INSTALL [for load], LOAD, or DELETE command returns an error status word, a required ELF cannot be loaded, and:

- If the recovery procedure has not yet been entered (i.e. the error was received during the first Loading Phase during which new ELF version(s) are being loaded), then the RAS should send one or several command scripts in order to:
  - Start the recovery procedure (with a MANAGE ELF UPGRADE [recovery] command);
  - Reload the old ELF version(s);
  - Re-enter the Restore Phase (with another MANAGE ELF UPGRADE [resume] command) and return to the initial state (i.e. old ELF version(s) and Application(s) in place).
- If the upgrade session has already entered the recovery procedure (i.e. the error was received during the second Loading Phase during which old ELF version(s) are being reloaded), then the RAS should send a MANAGE ELF UPGRADE [abort] command in order to fully abort the upgrade session.

See also section C.6, “About the ELF Upgrade Session’s Recovery Procedure”.

## C.6 About the ELF Upgrade Session’s Recovery Procedure

The success of the Recovery Procedure depends on the capabilities of the Application(s) involved in the upgrade session, which should have been developed to support the Recovery Procedure and should be able to re-integrate the instance data they saved during the Saving Phase. An Application that does not support recovery would throw an exception during the Recovery Procedure, leading to the full abortion of the upgrade session and full deletion of all the Applications involved in the upgrade session. The RAS should be aware of whether each Application supports recovery.

If the RAS knows that at least one Application involved in the upgrade session does not support the recovery procedure, it may send a MANAGE ELF UPGRADE [abort] command directly to fully abort the upgrade session instead of trying to enter or complete the recovery procedure.