Demoney: A Demonstrative Electronic Purse
— Card Specification —

Date: November 22, 2002
Authors: Renaud Marlet, Cédric Mesnil (Trusted Logic)
Classification: Public
Number: SECSAFE-TL-007
Version: 0.8
Status: Draft

Abstract. This document provides the specification of *Demoney*, a basic electronic purse for smart cards. The purse can be debited from a terminal in a store to pay a purchase, and credited at an ATM, from cash or withdrawing from a bank account. Demoney can also communicate with other applets on the card, e.g., to automatically award points on a loyalty plan when making a purchase in a store.

Although Demoney is too simple to be used as a real electronic purse, it is realistic enough to be representative of typical features found in smart card applications requiring security, such as banking applications. In particular, it involves authentication protocols and secure messaging, based on secret keys and challenges. Demoney is actually designed (and copyrighted) to be used as a demonstrative example, e.g., for illustrating security analyses, automatic test generators, program transformers, etc.

LEGAL NOTICE

COPYRIGHT NOTICE

DISCLAIMER OF WARRANTY

Contents

List of Tables

1 Introduction

This document provides the specification of *Demoney*, a demonstrative electronic purse for smart cards. This specification only concerns the part of the application that runs on a card, as opposed to the part of the application that runs on a terminal (card acceptance device) and provides an interface with the card holder.

1.1 Purpose

Demoney is very basic compared to a real electronic purse such as Moneo or CEP. There are less parameters and less features. As a matter of fact, Demoney is kept simple to be easily understandable. Still, it is realistic enough to be considered representative of actual applications in the banking domain. In particular, it fully addresses security issues regarding protocol attacks. However, it does not address other domain-specific issues such as non-repudiation. It does not either address more functional issues such as time stamp encoding or integer arithmetic — numbers that may be large (or precise, e.g., in the case of a balance with cents) are often encoded, e.g., in BCD format. The purpose of Demoney is actually only to illustrate typical features found in smart card applications requiring security.

Trusted Logic also provides a Java Card (JC) implementation of this specification. Like the specification, this implementation is kept simple to be easily understandable. In particular, it is not optimized for size nor heavily protected against hardware attacks. Still, it is representative of common code patterns found in real smart card applications.

Both the specification and the implementation of Demoney are to be used as the basis of demonstrative examples, typically for benchmarking program analysis, test generation, program transformation, etc. The specification may also be used as an example for a design environment. To this end, both the specification and the implementation can be modified, for instance to enforce or to loosen given security policies. These modifications may thus introduce intentional flaws, either in the security or in the correctness of the implementation with respect to the specification. Demoney is bound to evolve anyway, to adapt to new needs.

1.2 Copyright and Related Issues

The specification and the implementation of Demoney are not commercial products, neither as effective smart card applications nor as demonstrative objects. They are made publicly available. However, Trusted Logic holds rights on them. The right concerning this specification document is stated on page 1. The general idea is the following.

- No part of the present specification nor of the corresponding implementation (Java Card source code as well as any compiled or transformed code) can be used for commercial purposes without prior agreement from Trusted Logic.

- Any part of this specification and of the corresponding implementation can be quoted on any kind of medium for non-commercial purposes on the condition that Trusted Logic be mentioned as being the author and copyright holder.

- Both the specification document and the corresponding implementation can be modified on the condition that their copyright notices be kept unaltered, and that the nature of the modification be clearly stated in the document and/or in the code.

- Trusted Logic S.A. makes no representations or warranties, either expressed or implied, except to the extent that such disclaimers are held to be legally invalid, about the suitability of the software or of the related documentation, including but not limited to the implied warranty of merchantability, fitness for a particular purpose or non-infringement. In no event shall Trusted Logic S.A. be liable for any damages suffered by the licensee as a result of using, modifying or distributing this software or its derivatives.

In particular, the implementation has not been throughly tested as other commercial applications developped at Trusted Logic.

1.3 Specification Variants

Since Demoney aims at being representative of existing applications, it has also been designed to be compatible with existing standards, namely ISO / IEC 7816 and Open Platform. Open Platform (OP) and Visa Open Platform (VOP, a refinement of OP defined by Visa) are presently used as the basis for many cards and applications, in particular in the banking domain. This document actually defines three specification variants:

- DEMONEY-STAND-ALONE: All security issues are addressed without the need for any support from an underlying Open Platform system.
- DEMONEY-OP-2.0: Part of the security is addressed by Open Platform version 2.0. (It is not possible for all security aspects to be handled by OP 2.0, see §2.4 for details.)
- DEMONEY-OP-2.1: All security issues are addressed by Open Platform version 2.1.

However, these three variants have been designed to have as little differences as possible. More details about OP, VOP and the different specification variants are provided in Section 2.4.

1.4 Related Demonstrative Applications

A few other demonstrative applications have been made publicly available.

1.4.1 JavaPurse

Sun Microsystems Inc. (tm) also provides the implementation of a simple electronic purse, called JavaPurse [Sun01b]. JavaPurse can communicate with a loyalty applet called JavaLoyalty.

As Demoney, JavaPurse was developed to be an example. However, JavaPurse is less realistic and has more copyrights constraints than Demoney. More precisely, JavaPurse differs from Demoney in the following points.

- There is no up-to-date public specification of JavaPurse [Sun98].
- JavaPurse has little security features; in particular, it uses no cryptography at all.
- JavaPurse does not use any support from Open Platform.
- The copyright on JavaPurse is more strict. In particular, it does not allow code modification.

An even simpler purse is also made available by Sun: Wallet [Sun01c].

1.4.2 PACAP Purse and Loyalty

PACAP was a project involving Gemplus and ONERA (CERT, Toulouse) between January 1999 and December 2000 [Wie00]. The goal of the project was to provide techniques and tools enabling a smart card issuer to verify that a new applet securely interacts with already downloaded applets [BCM+00].

Two applets have been developped to provide a case study for the research: a purse and a loyalty application [BMGL01]. These applets are publicly available (under licence terms) on the Gemplus web site [Gem].

The PACAP purse and Demoney are similar in many respects, especially with respect to security. Still, there are differences, both regarding the specification and the Java Card implementation.

- The PACAP purse is more realistic regarding some features that are not related to security: it specifies a time and date format; it supports "floating poiint" numbers to model cents; it support an expiration date, an exchange rate, and a currency table. It also features (limited) debit and credit without authorization, and logging of errors and transaction certificate.
- The PACAP purse implementation is bigger than Demoney: it is about 30K (which makes it an *extremely* large applet) whereas Demoney is in the range 5-10K.
- The PACAP purse uses Open Platform only to set the state of the application and to get access to a global PIN. In that respect, it is closer to DEMONEY-STAND-ALONE rather than to DEMONEY-OP-2.0. It does not use the secure channel offered by Open Platform.
- Various models of PACAP are available : UML, SDL, IF, PVS [Gem]. Likewise, the development of models of Demoney is planned in other projects (UML-like model and input specification to a test generator).

As opposed to JavaPurse, the code of the PACAP purse can be modified.

1.5 Prerequisites

Although it is designed after security mechanisms defined in Open Platform, the DEMONEY-STAND-ALONE variant is fully specified in this document. On the contrary, the specifications of DEMONEY-OP-2.0 and DEMONEY-OP-2.1 are not self-contained; they rely on support defined in the Open Platform card specification and not always recalled in this document (see §2.4).

A reader not familiar with secure channels and in particular with secure channels *à la* OP should first read Appendix A.

1.6 Organization of the Document

The document is organized as follows. Section 2 provides an overview of the application, including security issues. Section 3 describes the application data. Section 4 presents the various states of the application. Section 5 details the different commands supported by Demoney. Appendix A details the security protocols and mechanisms used in Demoney. Appendix B provides additional specifications in the case of a Java Card implementation.

2 Application Overview

This section provides an overview of Demoney. It describes its features, discusses security issues and explains the design choices.

2.1 An Electronic Purse

Demoney is a basic electronic purse for a given, fixed currency. The purse can be debited from a terminal in a store to pay a purchase, and credited at an ATM, from cash or withdrawing from a bank

account. Demoney can also communicate with other applets on the card, e.g., to automatically award points on a loyalty plan when making a purchase in given stores.

For security reasons (see §2.2), debiting and crediting the purse requires the POS (Point Of Service or Point of Sale) to be authenticated as an authorized CAD (Card Acceptance Device). There are basically four different and increasing permission levels for operating the purse, corresponding to four different capabilities:

1. reading public information from the purse, such as the current balance,
2. debiting the purse,
3. crediting the purse (from cash or, possibly, via a bank account withdrawal),
4. performing administrative operations.

These four *access levels* actually correspond to four different kinds of terminals from which the purse can be operated:

1. any CAD able to select an application and ask for public data, e.g., PDA, GSM,
2. a terminal in a store, e.g., for debiting the purse as a purchase payment,
3. a bank terminal (ATM), e.g., for crediting the purse from cash or from a bank account,
4. an administration terminal in a bank, e.g., for updating configuration parameters.

Besides, crediting from a bank account requires PIN verification: the terminal is then assumed connected to the bank of the user with appropriate ability and authorization to perform the bank account withdrawal associated to the purse credit.

The general rule is that the balance of the purse must always stay positive. There are also ceilings concerning the maximum balance, the maximum debit amount in a transaction[1], and the maximum number of transactions. Besides, Demoney maintains a log of the most recent transactions. Each log record includes the purse transaction number, the currency, the transaction amount (debit or credit), the new balance, and transaction context (company operating the POS, POS identifier, POS transaction number, date and time).

Public data such as the current balance and the log file can be read without authentication or identification. On the contrary, setting internal parameters such as the maximum balance require the terminal to be authenticated as an administration terminal.

If the AID of a *purchase agent* has been given to a Java Card implementation of Demoney, the corresponding applet on the card (if any) is notified each time a purchase is made. This applet can be for instance a loyalty application awarding points for purchases made in given stores. In reality, it is unlikely for an electronic purse to directly communicate with a loyalty plan. This feature has been added to Demoney only as an example of applet communication. In the case of a Java Card implementation, this translates into using shareable interface objects.

Command Summary. After Demoney has been installed (i.e., when a Demoney instance has been created, given installation parameters and registered in the underlying system), the accepted commands are the following:

- STORE DATA: personalize the application,

[1]The word *transaction* is used with (at least) two different meanings. In the context of programming, a transaction can be an atomic update of a set of persistent data: either all data are updated, or none are. In the banking domain, transferring some amount of money from one account to another is also called a transaction. In this specification, “transaction” is only used in the latter sense. Still, it is likely that a programmatic transaction be used in an actual implementation of Demoney to guarantee the consistency of state updates.

- SELECT: select the application,
- INITIALIZE UPDATE: initialize a mutual authentication,
- EXTERNAL AUTHENTICATE: complete a mutual authentication,
- PUT KEY: store keys,
- PIN CHANGE / UNBLOCK: change the value of the PIN or unblock it,
- VERIFY PIN: verify the user PIN,
- INITIALIZE TRANSACTION: initialize a purse debit or credit (from cash or bank account),
- COMPLETE TRANSACTION: complete the transaction,
- GET DATA: read application data (besides log file),
- PUT DATA: update configuration parameters,
- READ RECORD: read a log file record.

Some commands are not available in all variants of Demoney. Or more precisely, they may have to be addressed to an OP secuiry domain rather than to the application itself. See Section 5 for details.

2.2 Security Issues

This subsection provides a rationale concerning security in the design of Demoney. It summarily describes the security issues: objectives, policies, and mechanisms. Security at the implementation level is also discussed.

Security Objectives. The security objectives in Demoney are threefold:

(1) No money can be created.

(2) It is difficult to accidentally destroy money.

(3) Only the owner of the bank account (if any) can debit the account to credit the purse.

Note that it is not possible to simultaneously prevent money to be created and to be destroyed. Letting money destruction be possible generally is not a problem because it usually can happen only in limited cases because of a material failure (accidental, or intentional, to attack the application); there is no protection here against voluntary money destruction[2].

Security Policies. These three security objectives are achieved using different security policies.

(1) To prevent money from being created,
   - Both the card and the terminal must authenticate themselves to each others. It must not be possible to forge an identity by replaying previous authentication messages.
   - Credit commands must be authenticated as originating from the terminal. It must not be possible to forge nor to replay them.
   - When crediting the purse from cash, money is collected before the purse is credited.
   - When crediting the purse from a bank account, the account is debited before the purse is credited.
   - When making a purchase, the POS gets a inimitable certificate of the purse debit.

(2) To make money destruction difficult,

[2] Just as anybody can burn a bank note — although it is illegal.

- The purse balance is decremented only if the debit is possible.
- The purse is first asked if a credit is possible before the bank account (if any) is debited or cash money is accepted. The purse credit is then confirmed at once.

Money can be destroyed only if an operation is interrupted before being completed. The "window" within which an interruption does destroy money is made as small as possible.

(3) To control the debiting of the associated bank account (if any),

- A PIN code must be entered to identify the card bearer. Only a limited number of unsuccessful identification attempts are possible.
- When crediting from a bank account, the POS gets a (temporary) inimitable certificate that a correct PIN code has been previously entered during the card session.

An additional policy, independent of Demoney, could be to limit the number and/or amount of weekly or monthly bank account withdrawal. This policy, that is to be set up at the bank level, limits the loss in case the PIN code is disclosed.

There are also limitations to transactions that are set at personalization time (and can be reconfigured later on from an administration terminal). These limitations make the exploitation of flaws more difficult: there is a maximum balance, a maximum debit amount, and a maximum transaction number.

Moreover, Demoney can only perform a limited absolute number of transactions. When the absolute maximum number of transactions is reached, the purse becomes inoperative. The purse instance is then to be deleted from the card (and possibly reinstalled if needed).

Note that Demoney is just like any ordinary, non-electronic purse: anybody finding the purse "on the ground" can empty it, i.e., use the present electronic money to make purchases, and refill it using cash money. But the bank account (if any) of the purse owner cannot be debited to refill the purse.

Security Mechanisms. Secure communications in Demoney are provided by *secure channels*, as described in Appendix A. Different mechanisms are used:

- Critical commands (debit, credit and administrative operations) can only be performed after mutual authentication of both the terminal and the card. Mutual authentication is based on shared secret keys and challenges (random numbers and counters). There are different keys, depending on the access level required for the operation. The challenges guarantee that the authentication messages cannot be replayed in another secure channel session.
- A signature guarantees not only the origin of critical messages (commands and responses) but also their integrity and their sequencing. For a signature not to be forged, it relies on a shared secret key. To prevent signed messages from being replayed:
  - Signatures are actually made with session keys, that are based on the shared secret keys and challenges. A session key is only valid for this secure channel session, i.e., between the opening of a secure channel and its closing. This prevents replays across sessions.
  - Signing a message depends on the value of the signature of the preceding signed message. This ensures the sequencing of messages and prevents replays within the session.

  Using session keys, that are temporary, also is a protection against attacks by cryptanalysis, that try to determine the value of keys by looking at a history of signed (or crypted) messages.
- Secret information is not observable: keys and PIN data are transmitted in crypted form.
- Crediting from a bank account additionally requires bearer identification, i.e., checking a secret code (PIN) recorded on the card.

Note that both command messages and response messages can be considered as critical and require signature.

Security at the Implementation Level. The security expressed above is only logical. It concerns the communication protocols and channels.

This specification document contains only few recommendations regarding protections against hardware attacks. These recommendations concern the atomicity of persistent data update, to ensure that the application always is in a consistent state. An actual implementation is free to also make the lifetime of secret data in plaintext as small as possible (to prevent observation) and to include redundant computations, time shifting and other kinds of countermeasures.

This specification does not mention any protection against software attacks either. In the case of a Java Card implementation, this is less of an issue as JC provides basic application isolation.

2.3 ISO 7816 Support

ISO / IEC 7816 is a standard for contact smart cards. Part 4 of the ISO 7816 specification defines a set of commands across all industries to provide access, security and transmission of card data. Within this basic kernel, for example, are commands to read, write and update records stored on the card. Some error codes are also normalized (see §5.2).

Commands. Besides the logic and format of messages, Demoney borrows from the following ISO 7816 standard commands (possibly via OP and VOP): SELECT FILE (actually called SELECT in Demoney), VERIFY (actually called VERIFY PIN in Demoney), GET DATA, PUT DATA, and READ RECORD. More precisely, the above commands are part of the commands accepted by Demoney; they are specialized to the data processed and stored into the purse.

TLV and LV. This specification document mentions TLV and LV formats. TLV is a specialization of the SIMPLE-TLV format described in the ISO 7816-4 norm. A TLV data object consists of 2 or 3 consecutive fields:

- The tag field T consists of 1 byte encoding a number from 1 to 254.
- The length field L consists of 1 byte encoding a number from 0 to 254.
- If L in not null, then the value field V consists of L consecutive bytes. If L is null, then the data object is empty: there is no value field.

The LV format is a TLV format without tag field.

2.4 Open Platform Support

As the specification of Open Platform (OP) puts it, OP is “a flexible and powerful specification for card issuers to create multi-application chip card systems” [Glo01]. Amongst other things, OP offers a support for card, key and PIN management. Open Platform is defined by the Global Platform organization (see www.globalplatform.org).

Visa Open Platform (VOP) is a refinement of OP defined by Visa [VIS00b]. Most OP implementation today actually are VOP implementations. (V)OP is presently used as the basis for many cards and applications, in particular in the banking domain.

Different OP Versions. There basically exist two different versions of Open Platform: OP 2.0 and OP 2.1. Whereas there is a single specification for OP 2.1 [Glo01], there are actually two revisions of version 2.0, and thus three variants: OP 2.0 [VIS99a], OP 2.0.1 [VIS99b], and OP 2.0.1' [VIS00a]. Since there are no significant differences between versions 2.0, 2.0.1 and 2.0.1' as far as Demoney is concerned, they are all named OP 2.0 in this document[3]. There are however important differences[4] between OP 2.0 and OP 2.1. In particular, these two versions offer different kinds of services regarding secure communication channels (see also Appendix A):

- *OP 2.0* offers secure channels consisting of mutual authentication as well as APDU command signature verification and possibly decryption. However, it does not offer any support for signing or encrypting response messages. In practice, OP 2.0 secure channels are thus only used to secure personalization (although they could also be used to sign and possibly decrypt command messages after personalization). Most OP 2.0 applications manage they own operational keys, independently of OP.
- In addition to the features of OP 2.0 secure channels, *OP 2.1* also offers support for signing response messages and for encrypting data using secure channel keys. It thus becomes possible for applications not to manage any key and to let OP perform all key treatments. OP 2.1 secure channels can thus be used not only for personalization but also for the regular processing of operational command and response messages, after personalization.

However, OP 2.1 is not just an extension of OP 2.0; there are inconsistencies. As a matter of fact, two *secure channel protocols* (SCP) are defined in OP 2.1:

- *SCP01* is intended to be compatible with OP 2.0.
- *SCP02* is a redesign of the secure channels of OP 2.0, offering not only response signature and data encryption but also an implicit mode for initiating secure channels (not used in the OP 2.1 variant of Demoney).

A security domain in OP 2.1 can be created as supporting either SCP01 or SCP02. (But the SCP cannot be changed later on.) In this document, when we mention OP 2.1, we understand SCP02 with explicit secure channels.

Demoney Variants. OP defines card management and centralize resources for applications, such as keys and PIN. This support cannot be emulated in a non-OP implementation without reimplementing much of OP. Since there are also differences between OP 2.0 and OP 2.1, this document actually specifies three variants of Demoney, corresponding to the following cases:

- DEMONEY-STAND-ALONE: All security (secure messaging and PIN) is defined without any support from an underlying Open Platform system. The keys and PIN used by Demoney are thus explicitly handled by the application. An implementation of this specification variant does not have to rely on any OP API.
- DEMONEY-OP-2.0: The keys used for personalization as well as the PIN are handled by Open Platform version 2.0. The other operational keys used by Demoney are explicitly handled by the application. An implementation of this variant has to call the OP 2.0 API.

[3] At the time this document is written, OP 2.1 [Glo01] is very recent; revisions may also be expected in the future to clarify some ambiguities.

[4] We only mention here differences regarding the specification. There are also major differences in the corresponding Java Card API. As a matter of fact, there is not a single common method in both APIs. The Java Card implementation of the different variants of Demoney thus cannot share any code related to the OP API.

- DEMONEY-OP-2.1: All security (secure messaging and PIN) is handled by Open Platform version 2.1 (with secure channel protocol SCP02). The keys and PIN used by Demoney are assumed to be already present on the card. An implementation of this specification variant has to call the OP 2.1 API.

To reduce differences among the variants, the secure messaging used by DEMONEY-STAND-ALONE follows the model defined in Open Platform 2.0, including cryptogram computation and verification as well as session key generation. See Appendix A for details.

Commands. To reduce variability, all Demoney variants also share commands that are borrowed from Open Platform: INITIALIZE UPDATE, EXTERNAL AUTHENTICATE, STORE DATA, SELECT. Commands PUT DATA and PIN CHANGE / UNBLOCK are borrowed from VOP. There are slight variations though, depending of the variant, that are detailed in Section 5. Installation is also inspired by OP.

Secure Channels and Key Sets. Secure messaging in Demoney is based on the Open Platform model. Communication operate through what is called a *secure channel*, that is a set of protocols and algorithms to guarantee secure messaging.

In short, a secure channel is opened after authentication of the two parties. It relies on shared static keys, that are used to generate session keys. Several static keys, forming a *key set* are actually involved: a message encryption key (also used for authentication), a signature key, and a key used to encrypt sensitive data, such as other secret or private keys. Secure channels are more thoroughly detailed in Appendix A.

2.5 Notations

The notations used in this document are the following.

- Hexadecimal values (nibbles, bytes and longer words) are written between single quotes, e.g., '2', '3F', '6A 86'.
- Lengths mentioned in tables (e.g., format of command messages) indicate a number of bytes.

Data format use the following terminology.

- *var.* indicates a variable quantity or format, defined in the text, e.g., length of a data field.
- *binary* indicates binary data with no structure, e.g., key data, cryptograms.
- *unspecified* indicates an unspecified format, e.g., date and time stamp. The application only needs to carry around the data, and possibly store it, without knowing how to interpret it.

Unless otherwise stated, integer data format are as follows.

- A *1-byte unsigned integer* is an integer in the range 0 to 255.
- *1-byte positive signed integer* is an integer in the range 0 to 127.
- A *2-byte unsigned integer* is an integer in the range 0 to 65535.
- A *2-byte signed integer* is an integer in the range -32768 to 32767.
- A *2-byte positive signed integer* is an integer in the range 0 to 32767.

All integers communicated between the CAD and the application are little endian (high order byte first).

3 Application Data

This section describes the various kinds of data stored in Demoney: constant data, persistent data and session data.

3.1 Constant Data

A few data are hard-coded in the Demoney specification:

- Demoney specifies the possible range of application data. For example, the PIN size (number of digits) must be in the range 4–12; the number of records in the log file must be in the range 1–50, etc. We do not provide in this section a complete list of valid ranges for the different parameters and variables; these ranges are given along with each data definition.
- The DEMONEY-STAND-ALONE variant also includes the constant, predefined, private, key data of a *bootstrap key set* [3 $\times$ 16 bytes]. These key data are used to build up, at installation time, the initial administration key set to be used at personalization time. These data somehow "bootstraps security" in a plain (non OP) system. The actual value of these key data is not part of this specification document.

The initial values of the state variables and log file, as well as the default value of the card and application session data can also be considered as constant data.

3.2 Persistent Data

Persistent data live from the creation of a Demoney instance up to the deletion of the instance from the card. Depending on the specification variant, security data (keys and PIN) are maintained by Demoney itself or by an OP underlying system.

As described below, persistent data maintained by Demoney are configuration parameters (§3.2.1), state variables (§3.2.2) and the log file (§3.2.3). These persistent data have to satisfy consistency conditions for Demoney to be in a correct state (§3.2.4).

3.2.1 Configuration Parameters

Configuration parameters are given at installation and personalization time.

Installation Parameters. *Installation parameters* are available or are provided at installation time. They consist of:

- the number of records in the log file,
- the PIN try limit,
- key diversification data (see §5.4).

These parameters are provided as *installation data*, i.e., as *application-specific parameters* of the installation command (see §5.4). They cannot be redefined.

Installation parameters also include a *personalization key set* to secure personalization (see §5.5). This security data is kept and operated by different entities depending on the specification variant:

- DEMONEY-STAND-ALONE: The personalization key set is handled by the application itself. It is generated from the predefined private *bootstrap key set* (see §3.1) and public diversification data provided as installation data (see §5.4). The personalization key set is actually treated as the *initial administration key set*, to be redefined at personalization time.

- DEMONEY-OP-2.0: The personalization key set is handled by Open Platform, independently of the application and installation data.
- DEMONEY-OP-2.1: The personalization key set is handled by Open Platform, independently of the application and installation data.

The personalization key set is private, it cannot be retrieved.

Personalization Parameters. The *personalization parameters* are provided at personalization time. They include:

- the purse identifier,
- the purse currency,
- the maximum transaction number,
- the maximum debit amount,
- bank information (optional and private),
- the AID of a purchase agent (optional).

These parameters are set using command STORE DATA, at access level *ADMIN*. Apart from the purse identifier, they can read and updated at any time after personalization from an administration terminal (commands GET DATA and PUT DATA). These parameters are associated with tags used for TLV-reading and -writing, as defined in Table 1.

Tag	Len.	Value		Format
1	4	Purse identifier	(1)	unspecified
2	2	Currency	(2)	unspecified
3	2	Maximum transaction number		unsigned integer
4	2	Maximum balance		positive signed integer
5	2	Maximum debit amount		positive signed integer
6	0 or 16	Bank information	(3)	unspecified
7	0 or 5–16	AID of a purchase agent	(3)	byte array

(1) The purse identifier can only be set once, at personalization time.

(2) The current balance must be null for the currency to be updatable.

(3) This optional parameter can be set and unset. It does not determine personalization.

Table 1: Personalization Parameters (except Keys and PIN)

The format of the purse identifier is not specified in this document. This identifier is only stored at personalization time (command STORE DATA) and retrieved on selection (command SELECT) and when reading public information (command GET DATA).

The format of the currency is not specified either. It is enough to make sure that the purse currency defined as a configuration parameter (commands STORE DATA and PUT DATA) exactly matches currency used in transactions (see command INITIALIZE TRANSACTION).

The format of the bank information, defined as a configuration parameter (commands STORE DATA and PUT DATA), is not specified either. It identifies a bank account to debit for loading the purse. The interpretation of this information, typically retrieved by command INITIALIZE TRANSACTION [credit from bank account], is left to the terminal.

Besides these operational data, personalization parameters also include private security parameters, that are set by specific commands:

- the *debit key set*, used to secure debit transactions (the DEK of this key set is unused in this version of Demoney),
- the *credit key set*, used to secure credit transactions,
- the *administration key set*, used to secure parameter updates (including PIN and keys) as well as PIN unblocking,
- a PIN (optional).

These security parameters are set using commands PUT KEY and PIN CHANGE / UNBLOCK. They can be updated at any time after personalization from an administration terminal. Security parameters are kept and operated by different entities depending on the specification variant:

- DEMONEY-STAND-ALONE: Keys and PIN are stored and operated by the application itself. The PIN is specific to the application.
- DEMONEY-OP-2.0: The keys are stored and operated by the application itself. The PIN is handled by Open Platform; it is global to the card.
- DEMONEY-OP-2.1: The keys and PIN are all handled by Open Platform. The PIN is global to the card.

The keys and PIN are private parameters. They are provided as ciphertext and cannot be retrieved.

Demoney maintains information to know which parameters have been given a value or are considered to be undefined. Demoney can be considered personalized only when the above mandatory parameters are defined. These parameters cannot later be unset, contrary to optional parameters that can be defined and undefined at any time. If an optional parameter is not defined, the purse is globally operational but cannot perform the specific corresponding feature (e.g., credit from a bank account, purse agent notification).

All parameters maintained by the application itself are considered undefined initially and have a null value. Parameters that are maintained by Open Platform (depending on the specification variant) are always considered defined.

3.2.2 State Variables

Like configuration parameters, *state variables* are given a value at personalization time. Unlike configuration parameters, this value is implicit. State variables consists of:

- the current transaction number (initially set to 0),
- the current balance (initially set to 0),
- the PIN presentation counter (initially set to the maximum number of tries),
- the administrative status (see §3.4.1).

State variables have a predefined, initial value and are updated as side-effects of performing commands, e.g., when checking the PIN, when processing debit and credit transactions. These variables are associated with tags used for TLV-reading, as defined in Table 2.

State variables are not to be confused with operational states, that corresponds to logical states of the application (see §4). The administrative status provides a view on some of those states though.

Tag	Len.	Value	Format
14	2	Transaction number	unsigned integer
15	2	Balance	positive signed integer
16	1	PIN presentation counter	positive signed integer
17	1	Administrative status	see Table 6

Table 2: State Variables

3.2.3 Log File

The *log file* records the most recent transactions. It is a cyclic file: appending a new record deletes the older one. The last (newest) record entry is number 1, preceding entry is number 2, and so on. The log file size, i.e., the number of log records, is fixed; it is given at installation time (see §5.4).

Log records are detailed in Table 3. They can be read using command READ RECORD. A log record contains information concerning the purse update. It also contains the external context of the transaction, described in Table 4.

Len.	Value	Format
2	Purse transaction number	unsigned integer
2	Transaction currency	unspecified
2	Transaction amount	signed integer
2	New balance	positive signed integer
16	Transaction context	see Table 4

Table 3: Transaction Log Record

Len.	Value	Format
4	Identifier of company operating the POS	unspecified
4	POS terminal identifier	unspecified
4	POS transaction number	unspecified
4	Date and time stamp	unspecified

Table 4: Transaction Context

In a transaction context, the identifier of the company operating the POS, and possibly the POS terminal identifier, can be used by a loyalty application defined as a purchase agent to grant points when a purchase is made in given stores. The format of these identifiers is not specified in this document though, nor is that of the POS transaction number, and the date and time stamp.

The log file is a kind of structured state variable, in the above sense (see §3.2.2). It initially consists of records exclusively made of binary zeros. A null purse transaction number is enough to indicate a void record.

3.2.4 Update and Consistency Conditions

The application parameters cannot be given any value; there are consistency constraints. A correct configuration of persistent data is characterized as follows:

- All mandatory parameters are set (see §3.2.1).
- The transaction number is lesser than the maximum transaction number.
- The balance is positive.
- The balance is lesser than the maximum balance.
- The maximum debit amount is positive.

Besides consistency, updating parameters must satisfy the following constraints:

- The currency can only be updated if the balance is null.
- All updates of persistent data have to be performed atomically.

Atomicity guarantees that the above constraints are not broken at any time.

Note that the maximum transaction number and the current transaction number are 2-byte unsigned integers, in the range 0-65535. The maximum transaction number can be assigned any value provided it is greater or equal than the current transaction number. This is typically performed from an administration terminal when the transaction number has reached the configured maximum. The current transaction number can only increase. It is bounded by 65535. After that, the purse becomes inoperative.

3.3 Session Data

Session data are temporary. They consist of the following kinds of data, that are listed in decreasing lifetime order.

- *Card session data* live from the first application selection up to card reset. They have a default initial value.
- *Application session data* live from application selection up to deselection. They have a default initial value.
- *Secure channel session data* live from a successful EXTERNAL AUTHENTICATE command up to deselection or the next successful EXTERNAL AUTHENTICATE command. They do not have any default value.
- *Sequencing data* live between two commands that must be successive, i.e., between INITIALIZE UPDATE and EXTERNAL AUTHENTICATE, and between INITIALIZE TRANSACTION and COMPLETE TRANSACTION. They do not have any default value.

The session data maintained by Demoney are given in Table 5. The indentation indicates data that exist (i.e., make sense) only when other some data have given values. For instance, session keys only makes sense (and are used) when the access level is different from *PUBLIC*.

As the session data is not observable from the outside of the application, the exact format is implementation-dependent. In particular, some flags may be implemented as conditions on other session data rather than explicit booleans or bits. For instance, testing the flag indicating a successful preceding INITIALIZE TRANSACTION may actually consist in checking that the tentative credit amount is strictly positive, a null value signifying that the preceding command was not a successful INITIALIZE TRANSACTION.

In addition, a Java Card implementation without OP support typically also requires objects that represent a signature generator and verifier, an encryption algorithm, and a random number generator.

Value		Lifetime
Current access level (default: *PUBLIC*)		application session
– Current security level	(1,2)	application session
– Message encryption/authentication session key [2-key triple DES]	(1,2)	secure channel
– C-MAC session key [2-key triple DES]	(1,2)	secure channel
– C-MAC of previous signed command message [8 bytes]	(1,2)	secure channel
– R-MAC of previous signed response message [8 bytes]	(1,2)	secure channel
– Currently open secure channel [1 byte]	(2)	secure channel
Successful preceding INITIALIZE UPDATE flag (default: false)		application session
– Tentative access level		next command
– Host challenge [8 bytes]	(1,2)	next command
– Card challenge [8 bytes]	(1,2)	next command
Successful preceding INITIALIZE TRANSACTION flag (default: false)		application session
– Tentative transaction amount [signed integer]		next command
PIN validation flag (default: false)	(1)	card session

(1) This variable is part of DEMONEY-STAND-ALONE.

(2) This variable is part of DEMONEY-OP-2.0.

Table 5: Session Data

3.4 Summary Data

Two summaries of the Demoney internal data are defined: an administrative status (§3.4.1) and an operational status (§3.4.2). These summaries are provided in the response message of some commands, such as SELECT.

3.4.1 Administrative Status

The *administrative status* is a set of flags indicating:

- which Demoney variant (see §2.4) is implemented by the application (this information actually is constant),
- the personalization state (i.e., personalized or not),
- whether the purse can be credited from a bank account (i.e., if both a PIN and bank information are defined),
- whether debiting the purse notifies a purchase agent (i.e., if the AID of a purchase agent is defined, if there exist an application with this AID on the card, and if it agrees to communicate with the purse), and
- whether the PIN has been validated. (This actually is not a persistent information; it belongs to the session data, see §3.3.)

The format of the administrative status is given in Table 6. This information is returned by the purse to the terminal on a SELECT command.

bit 8 … bit 1	Meaning
– – – – – 0 0 0	Application implements DEMONEY-STAND-ALONE
– – – – – 0 0 1	Application implements DEMONEY-OP-2.0
– – – – – 0 1 1	Application implements DEMONEY-OP-2.1
– – – 1 – – – –	Application is personalized
– – 1 – – – – –	Purse may be credited from a bank account
– 1 – – – – – –	Debiting the purse notifies a purchase agent
1 – – – – – – –	PIN has been validated

Table 6: Bit Coding of the Administrative Status

3.4.2 Operational Status

The *operational status* is a "user-friendly" summary of the application data. This information can be displayed to the card bearer and can be used by the POS before attempting a debit or credit transaction. It is described in Table 7, with the following definitions.

- The remaining number of transaction is the maximum number of transaction minus the current transaction number.
- The maximum allowed debit is the minimum of the current balance and the maximum debit amount specified as a configuration parameter.
- The maximum allowed credit is the maximum balance minus the current balance.

Len.	Value	Format
2	Remaining number of transactions	unsigned integer
2	Purse currency	unspecified
2	Current balance	positive signed integer
2	Maximum allowed debit	positive signed integer
2	Maximum allowed credit	positive signed integer

Table 7: Operational Status

This information is returned by the purse to the terminal on a SELECT command and after each successful transaction (command COMPLETE TRANSACTION).

4 Application States

This section describes the various states of the application: life cycle, access levels, security levels, personalization, command sequencing, security thresholds, optional features. Some commands are allowed only in given states; if a command is issued in an inappropriate state, an error is raised. See Section 5.2 for a list of standard errors.

4.1 Application Life Cycle

The life cycle of Demoney follows these stages:

- LOADED: The application code is loaded onto the card.

- INSTALLED: A specific instance of the application has been created and registered in the underlying system (e.g., the Java Card Runtime Environment). Some data structures may have been allocated and initialized using installation parameters.
- SELECTABLE: The application instance is selectable. Still, the only accepted operation at this point is personalization.
- PERSONALIZED: the application instance is personalized and made fully operational.

If the state PERSONALIZED is not managed by the underlying system, it has to be handled by the application itself (see §4.2). Specific Open Platform implementations might also include other states, such as LOCKED [VIS99a], but these are states that are handled by the card manager (if any), not the application itself.

4.2 Personalization

The constraints on the personalization state express the fact that the application must be personalized (i.e., configured) to be operational. Demoney is personalized if and only if:

- All mandatory configuration parameters mentioned in Table 1 are defined.
- DEMONEY-STAND-ALONE: the PIN is defined.
- DEMONEY-STAND-ALONE and DEMONEY-OP-2.0: all keys are defined.

Before personalization, the only accepted commands are:

- SELECT,
- STORE DATA,
- PUT KEY (not part of DEMONEY-OP-2.1),
- PIN CHANGE / UNBLOCK (only part of DEMONEY-STAND-ALONE),
- INITIALIZE UPDATE (not required in DEMONEY-OP-2.1),
- EXTERNAL AUTHENTICATE (not required in DEMONEY-OP-2.1),
- GET DATA.

The command INITIALIZE UPDATE is used to set the access level (see §4.3) to *ADMIN* before performing personalization. The command EXTERNAL AUTHENTICATE sets the security level (see §4.4).

After personalization, all commands are accepted but STORE DATA. There are no differences among the Demoney variants in that respect. However, depending on the variant, some of these commands are to be sent to the security domain of the application rather than to the application itself. See Section 5 for details.

Attempting to process a command in an inappropriate state with respect to application personalization raises error “Conditions of use not satisfied”.

4.3 Access Levels

Demoney operates with different access levels, which correspond to different authentications and permissions. There is a strict hierarchical chain of access levels: each level allows specific operations, and any operation available at a lower level is also available at an upper level. The different access levels, ordered with increasing permission (i.e., more allowed commands), are as follows.

1. *PUBLIC:* no authentication performed. Operating commands accepted at this level are:

- GET DATA,
- READ RECORD.

The other commands accepted at this level are:

- INITIALIZE UPDATE,
- EXTERNAL AUTHENTICATE,
- VERIFY PIN.

These commands are used to change the access level. *PUBLIC* is the initial access level, each time the application is newly selected.

2. *DEBIT:* authentication of a terminal allowing the purse to be unloaded, e.g., a terminal in a store. The additional operating commands accepted at this level are:
   - INITIALIZE TRANSACTION [debit].
   - COMPLETE TRANSACTION.

3. *CREDIT:* authentication of a terminal allowing the purse to be loaded from cash or from a bank account, e.g., an ATM. The additional operating command accepted at this level is:
   - INITIALIZE TRANSACTION [credit from cash],

   Crediting from a bank account requires the user PIN to be verified (see level *CREDIT-IDENT*).

4. *CREDIT-IDENT:* authentication of a terminal allowing the purse to be loaded from cash or bank account as well as PIN identification of the purse bearer. The additional operating command accepted in this state is:
   - INITIALIZE TRANSACTION [credit from bank].

5. *ADMIN:* authentication of an administration terminal, i.e., a specific terminal from the bank that issued the purse. The additional operating commands accepted at this level are:
   - PUT DATA,
   - PUT KEY (not part of DEMONEY-OP-2.1),
   - PIN CHANGE / UNBLOCK (only part of DEMONEY-STAND-ALONE),
   - STORE DATA (access level *PUBLIC* is enough for DEMONEY-OP-2.1, as security is granted in this case by the security domain).

   Variant DEMONEY-OP-2.1 does not require access level *ADMIN* for STORE DATA because the command is actually sent to the security domain through an OP secure channel rather than to the application.

Attempting to process a command with an inappropriate access level raises an error "Security status not satisfied" (see §5.2).

4.4 Security Levels

In addition to access levels, secure channels also have a *security level* attribute that indicates whether command and response messages must be signed. This security level is borrowed from Open Platform (see §A.2.1). In Demoney, the following messages must be signed:

- Personalization commands (C-MAC).
- Commands at any access level greater than *PUBLIC* (C-MAC).

- Commands and responses to INITIALIZE TRANSACTION and COMPLETE TRANSACTION (C-MAC and R-MAC).

Attempting to process a command with an inappropriate security level raises an error "Security status not satisfied" (see §5.2).

4.5 Command Sequencing

Some features of Demoney require sending two commands rather than one. Constraints concerning command sequencing express the fact that the first, *initiating command* must be immediately followed by the corresponding, second, *terminating command*:

- INITIALIZE UPDATE must be immediately followed by EXTERNAL AUTHENTICATE.
- INITIALIZE TRANSACTION must be immediately followed by COMPLETE TRANSACTION.

If a terminating command is not *immediately* preceded by a *successful* corresponding initiating command, the application raises the error "Conditions of use not satisfied".

4.6 Security and Operational Thresholds

Demoney contains various thresholds that address security and operational issues. When a threshold is reached, some commands becomes unavailable.

PIN try limit. If the number of invalid PIN presentations reaches the maximum try limit, the PIN is blocked: command VERIFY PIN always fails later on, with error "(PIN) authentication method blocked". The PIN can be unblocked using command PIN CHANGE / UNBLOCK.

Transaction Number Limit. If the current transaction number has reached the maximum transaction number, the commands INITIALIZE TRANSACTION and COMPLETE TRANSACTION are not accepted anymore: the application raises the error "Maximum number of transactions reached".

Debit Limit. Debiting the purse is accepted only if the balance stays positive or null and if the debit amount is not greater than the maximum debit amount. If debit is refused, error "Debit amount too high" is raised.

Credit Limit. Crediting the purse is accepted only if the balance stays lesser or equal to the maximum balance. If credit is refused, error "Credit amount too high" is raised.

4.7 Optional Features

Demoney can be configured with various options. If an optional feature that is not enabled is requested, an error may be raised.

Credit from Bank Account. Demoney can be configured to load the purse by withdrawing money from a bank account. PIN and bank information must be defined for the withdrawing capacity to be enabled. If not, a request to credit from a bank account raises the error "Function not supported".

Purchase Agent Notification. Demoney can be configured to notify an application that is also present on the card when completing a debit operation. The AID of this purchase agent application must be defined for this notification capacity to be enabled. However, no error is raised if no such AID is defined, if there is currently no application with this AID on the card, or if interacting with this application fails for any reason.

5 Commands

This section details the different commands accepted by Demoney. Installation is not treated as a standard command because it depends much on the underlying card system. Personalization also may depend on the underlying system. After a successful installation and personalization, the application can be selected and can receive operational commands.

5.1 Message Format

The format of commands and responses is defined according to the ISO 7816 norm. Demoney also borrows actual commands from ISO 7816 (see §2.3) and from Open Platform (see §2.4).

In APDU command headers, class code '94' is used for the commands that are specific to Demoney. (The third bit is set to indicate secure messaging commands, see norm ISO 7816.) The commands that are borrowed from ISO 7816 and OP use the corresponding class codes, i.e., '00' for ISO 7816, and '80' and '84' for OP.

The secure messaging used in Demoney consists in signing commands as well as some response messages (see Appendix A). A C-MAC (respectively, R-MAC) is appended at the end of the data field to sign a command (respectively, response) message.

5.2 Status Words

All response messages are terminated by a *status word* consisting of two bytes. Table 8 describes the general status words that may be returned by any command, as defined in the ISO / IEC 7816 norm (see §2.3) and in Open Platform (see §2.4).

Status words that are specific to Demoney are listed in Table 9. The meaning of these error codes[5] is as follows:

- "Maximum number of transactions reached" indicates that the purse transaction number has reached its maximum value and thus that the requested new transaction cannot be performed.
- "Currency error" indicates that the purse currency differs from the transaction (debit or credit) currency. The transaction cannot be performed. (There is no currency conversion.)
- "Credit amount too high" indicates that incrementing the purse balance with the requested credit amount would overflow the the maximum allowed balance. The credit transaction cannot be performed.
- "Debit amount too high" indicates that the debit amount is either greater than the maximum debit amount or greater than the current purse balance. The debit transaction cannot be performed.
- "Invalid transaction amount" indicates that the transaction (debit or credit) amount is negative or null. Only transactions with a strictly positive amount can be performed.

[5] These error codes are inspired by CEP.

SW1	SW2	Meaning
’63’	’00’	Authentication of host cryptogram failed
’63’	’Cx’	(PIN) Verification failure; x indicates the number of further allowed retries
’67’	’00’	Wrong length in Lc
’69’	’82’	Security status not satisfied
’69’	’83’	(PIN) authentication method blocked
’69’	’85’	Condition of use not satisfied
’6A’	’80’	Incorrect parameters in the data field
’6A’	’81’	Function not supported
’6A’	’83’	Record not found
’6A’	’85’	Lc inconsistent with TLV structure
’6A’	’86’	Incorrect parameters P1-P2
’6A’	’88’	Referenced data not found
’6C’	’xx’	Wrong length in Le; xx indicates the exact length
’6D’	’00’	Instruction code not supported or invalid
’6E’	’00’	Class not supported
’90’	’00’	Successful execution of the command

Table 8: Standard Status Words

SW1	SW2	Meaning
’91’	’02’	Maximum number of transactions reached
’94’	’01’	Currency error
’94’	’02’	Credit amount too high
’94’	’03’	Debit amount too high
’94’	’04’	Invalid transaction amount (must be strictly positive)

Table 9: Demoney-Specific Status Words

5.3 General Command Processing

All command processing in Demoney follows this general pattern:

- Check message signature, if any, and unwrap command.
- Check format of command header.
- Check that the command is valid in the current application state.
- Perform the command.
- Send the response message, if applicable.

The rationale for raising errors when processing a command is the following:

- Apart from the message signature, that is checked first although located at the end of a command message, errors are raised as soon as possible when scanning the APDU command message; the command header is checked before the data field.
- For a given piece of scanned data, syntax is checked before semantics, i.e., the expected value range is checked before checking the consistency of the value with respect to other data.
- Controlling accepted commands depending on state conditions (see §4) are checked in the fol-

lowing order: personalization done, command sequencing, optional feature, access level, security level, security and operational thresholds.

More precisely, command processing in Demoney is as follows.

1. If the class code is unknown (i.e., different from ’00’, ’80’, ’84’, ’94’), return error “Class not supported”.
2. If the class code indicates secure messaging (i.e., ’84’ or ’94’), unwrap the APDU:
   - If a secure channel is not currently open, return error “Conditions of use not satisfied”.
   - If Lc is lesser or equal to 8 (the MAC length), return error “Wrong length in Lc”.
   - If MAC verification fails, return error “Security status not satisfied”.
   - Otherwise remove the MAC from the data field: decrement Lc by 8.
3. If the class code does not indicate secure messaging (i.e., ’00’ or ’80’), close the currently open secure channel, if any.
4. If the instruction code is unknown (including the case when a known instruction is unavailable the given Demoney variant), return error “Instruction code not supported or invalid”.
5. If the values of the parameters P1 or P2 are not in the expected range, return error “Incorrect parameters P1-P2”.
6. If the data field length Lc (in the case the command has a non empty data field) is not in the expected range (possibly depending on P1 or P2), return error “Wrong length in Lc”.
7. If the maximum expected response length Le (in the case the command has non empty data field) is not in the expected range, return error “Wrong length in Le”.

These first steps form the general part of command processing that applies to all Demoney commands. This generic treatment is not recalled in the detailed description of each command processing, as provided in Section 5.

The following steps are specific to each command. We provide here the general pattern of treatment; each command is further detailed in the following subsections.

8. If the application is not in the appropriate personalization state (see §4.2), return error “Conditions of use not satisfied”.
9. If the application is not in the appropriate sequencing state (see §4.5), return error “Conditions of use not satisfied”.
10. If the values of the parameters P1 or P2 indicate a feature unavailable in the current application configuration, return error “Function not supported”.
11. If the current access level is too low for the command (see §4.3), return error “Security status not satisfied”.
12. If the current security level is not appropriate for the command (see §4.4), return error “Security status not satisfied”.
13. If the command corresponds to a debit or credit transaction and the maximum transaction number is reached (see §4.6), return error “Maximum number of transactions reached”.
14. If the contents of the data field is incorrect with respect to the present application state and data, return either an application-specific error message, “Security status not satisfied”, or “Incorrect parameters in the data field” if no specific error message apply (see §5.2). Checking the correctness of the data field may involve both syntax checking, e.g., in case of sequences of TLVs or LVs, and semantic checking, e.g., to check than the provided values are in the appropriate range and are consistent.

15. If the data field length Lc is incompatible with the contents of the data field, then if the data field represents TLVs return error "Lc inconsistent with TLV structure" otherwise return error "Incorrect parameters in the data field".
16. Perform the command. (This may raise other errors: failed authentication, etc.)

This processing applies to commands that are directly treated by Demoney. As for the installation command, that is partly treated by the underlying system, Demoney itself only raises error "Incorrect parameters in the data field".

5.4 Installation

Installation is the phase during which an application instance is built and registered in the underlying system. Some application data structures are also possibly allocated at this time.

Installation commands are actually addressed to the underlying system (e.g., Open Platform) rather than to Demoney, as an application instance does not exist yet. It precisely is the role of installation to create this instance and to register it. As a result, installation is very system-dependent (see below).

Enough dimensioning parameters are available at installation time for the allocation of *all* persistent data areas (see §3.2). The session data areas that are implemented with system support (e.g., transient arrays in Java Card) can also be prepared at installation time.

After a successful installation, the application can be made selectable (see §4.1). In an Open Platform setting, installing an application and making it selectable can be done in one or two commands. Still, the application only becomes operational after personalization.

DEMONEY-STAND-ALONE. Not making any assumption on the underlying system, this variant only specified the format of the installation data (see below). This include key diversification data to generate the personalization key set.

DEMONEY-OP-2.0 and DEMONEY-OP-2.1.

- The terminal sends the commands INITIALIZE UPDATE and EXTERNAL AUTHENTICATE to the security domain of the application. This performs mutual authentication and define the security level and the session keys of the secure channel. The actual security level and the associated static keys are not part of this specification.
- The terminal possibly sends a PUT KEY command to the security domain through the secure channel to define a personalization key set, if not already present on the card.
- The terminal sends an INSTALL [for install] command to the security domain through the secure channel, providing installation data, as defined in OP 2.0. (The application can also be made selectable at this time using command INSTALL [for make selectable].)

Installation Data. Installation data for all variants are described in tables 10 and 11. Installation data are provided as an ordered LV sequence. Application-specific parameters are provided as an ordered sequence of values, without lengths (that are fixed).

In the case of a Java Card implementation (with or without support from OP), this installation data is exactly the contents of the argument byte array of the `Applet.install` method.

Len.	Value	Format
5–16	Demoney instance AID	byte array
1	Application privileges (no privileges) (1)	imposed value '00'
1 or 50	Application-specific parameters (2)	see Table 11

(1) This item is not part of the DEMONEY-STAND-ALONE specification variant.

(2) Length is 50 for the DEMONEY-STAND-ALONE variant; it is 1 in the other cases.

Table 10: Installation Data (Ordered LV Data)

Len.	Value	Format
1	Number of records in the log file	1–50
1	Maximum number of times an incorrect PIN can be presented (3)	3–15
16	Diversification data for the initial administration S-ENC key (3)	binary
16	Diversification data for the initial administration S-MAC key (3)	binary
16	Diversification data for the initial administration DEK (3)	binary

(3) This item is only part of the DEMONEY-STAND-ALONE specification variant.

Table 11: Application-specific Parameters (Ordered Values)

Command Processing.

- If the installation data (lengths and values) are not in the expected range, or if the total length of the installation data is inconsistent with the individual lengths, return error "Incorrect parameters in the data field".
- Atomically:
  - Create a log file with the given log file size.
  - DEMONEY-STAND-ALONE only: Create a PIN with maximum size 12 digits and with the given try limit.
  - DEMONEY-STAND-ALONE only: Xor the key diversification data with the bootstrap key data (see §3.1) to form the key data of the initial administration key set, i.e., the personalization key set. Record this key set.
  - Considere that all configuration parameters are undefined (see §3.2.1). Note that:
    - DEMONEY-STAND-ALONE: it includes all key sets and PIN,
    - DEMONEY-OP-2.0: it includes all key sets but not the PIN (as the PIN defined in OP),
    - DEMONEY-OP-2.1: it does not include any key set nor PIN.
  - Initialize state variables (see §3.2.2): initialize the transaction number and the balance to zero; initialize the PIN try counter to its maximum value (DEMONEY-STAND-ALONE only).
  - Initialize the log file: initialize all records with binary zeros.
  - Register the application instance in the underlying system with the given instance AID.

An implementation may allocate at this time all data structures for persistent data and prepare session data areas. It may also delay those allocation until personalization, or even until the first use of the data structure.

Response Message. The data field of the response message is not present. This command may return a general condition error as listed in Table 8.

5.5 Personalization

Personalization sets all the configuration parameters that are required before the application can be operationally used (see §4.2). Some parameters are optional; the application can be personalized with or without:

- the capability for loading the purse from a bank account,
- a purchase agent to notify at each debit transaction.

If not enabled at personalization time (using command STORE DATA), these features can later be enabled (using command PUT DATA).

Personalizing a selectable application instance is partly system-dependent, which explains the different following cases. Still, in all cases, personalization is completed by a successful STORE DATA command and can only be performed once.

DEMONEY-STAND-ALONE.

- The terminal sends the commands INITIALIZE UPDATE [admin] and EXTERNAL AUTHENTICATE [C-MAC] to the application, indicating the personalization key set, i.e., the initial administration key set defined at installation time (see §5.4). This performs mutual authentication and defines the security level (command signature) and session keys of the secure channel. This also set the access level to *ADMIN*.
- Then the terminal sends personalization commands to the application: several PUT KEYs and PIN CHANGE / UNBLOCK, and then STORE DATA. It is up to the application to process the secure messaging: the message signature is checked and the data field of the command is decrypted; key data and PIN have to be further decrypted.

DEMONEY-OP-2.0.

- The terminal sends the commands INITIALIZE UPDATE [admin] and EXTERNAL AUTHENTICATE [C-MAC] to the application, indicating the personalization key set. This performs mutual authentication and defines the security level (command signature) and the session keys of the secure channel. This also set the access level to *ADMIN*.
- The terminal sends personalization commands to the application: several PUT KEYs and then STORE DATA. It is up to the application to process the secure messaging: the message signature is checked and the data field of the command is decrypted; secret data such as keys have to be further decrypted. The Open Platform Java Card API provides support to implement these operations (see §B.2).

The global PIN may also be personalized. (It is assumed to always be defined.) In VOP 2.0.1', setting the global PIN requires sending a PIN CHANGE / UNBLOCK command to the card manager in a secure channel.

DEMONEY-OP-2.1.

- The terminal sends the commands INITIALIZE UPDATE and EXTERNAL AUTHENTICATE to the security domain of the application. This performs mutual authentication and define the security level of the secure channel. The actual security level and the associated static keys are not part of this specification.
- The terminal sends PUT KEY commands to the security domain through the secure channel to define the debit, credit and administration key sets.
- The terminal then sends a command INSTALL [for personalization] to the security domain through the secure channel, indicating the AID of the application instance to personalize.
- Last, the terminal sends a command STORE DATA to the security domain through the secure channel. The secure messaging command is automatically filtered by the security domain: the message signature is checked (if any) and the data field of the command is decrypted (if applicable). The unwrapped data is then forwarded in plaintext to the application, for it to process the personalization data.
- Personalization is completed after a successful STORE DATA command.

The global PIN — actually called Cardholder Verification Method (CVM) in OP 2.1 — may also be personalized. (It is assumed to always be defined.)

5.6 STORE DATA

STORE DATA is the main command used to personalize a selectable application instance (see §5.5). It is a specialization of the OP 2.1 command with the same name. It can only be performed successfully once.

This command requires a secure channel to be open. In DEMONEY-STAND-ALONE and in DEMONEY-OP-2.0, the command is sent to the application. In DEMONEY-OP-2.1, the command is sent to the the security domain of the application.

Command Message. The format of the STORE DATA command message is given in Table 12. The configuration parameters are provided as an unordered sequence of TLVs. The tag and constraints concerning lengths and values are given in Table 1.

Field	Len.	Value	Meaning
CLA	1	’84’	OP command with secure messaging
INS	1	’E2’	STORE DATA
P1	1	’80’	Last block
P2	1	’00’	Block #0
Lc	1	*var.*	Length of the TLV sequence
Data	*var.*	see Table 1	Configuration parameter TLVs
Le	0	—	empty

Table 12: STORE DATA Command Message

Command Processing.

- DEMONEY-OP-2.1: Return error "Condition of use not satisfied". (This command must be sent to the security domain of the application, not to the application itself.)
- If the application is already personalized, return error "Condition of use not satisfied".
- DEMONEY-STAND-ALONE and DEMONEY-OP-2.0 only: If the access level is not *ADMIN*, return error "Security status not satisfied".
- Atomically:
  - While there are still some TLVs to decode (as indicated by the data field length Lc),
    - If decoding the TLV requires reading more bytes than indicated by the data field length Lc, return error "Lc inconsistent with TLV structure".
    - If tag T is unknown (i.e., not in Tables 1 or 2), return error "Incorrect parameters in the data field".
    - If tag T is known but corresponds to a state variable (Table 2) rather than to a configuration parameter (Table 1), return error "Function not supported".
    - If length L is not in the expected range for the parameter indicated by T (including possible or impossible value zero), return error "Incorrect parameters in the data field".
    - If length L is null, consider the value as undefined.
    - If length L is not null,
      - If value V is not in the expected range, return error "Incorrect parameters in the data field".
      - Consider the value as defined and store it in persistent memory.
  - If mandatory configuration parameters are not all defined, return error "Incorrect parameters in the data field".
  - If the final configuration of persistent data is inconsistent (see §3.2.4), return error "Incorrect parameters in the data field".

If no error is returned, personalization is considered successful and no subsequent STORE DATA command can be performed (the command returns error "Condition of use not satisfied").

Note: given that the purse identifier is mandatory and cannot be unset, the application is personalized if and only if the purse identifier is defined.

Response Message. The data field of the response message is not present. This command may return a general condition error as listed in Table 8.

5.7 SELECT

The SELECT command selects a given instance of the application. It is a specialization of the SELECT command in Open Platform, which itself is a specialization of the SELECT FILE command in the ISO 7816 norm. The response to the SELECT command is a statement of the purse identifier, state, contents and operating limitations.

Command Message. The format of the SELECT command message is given in Table 13.

Command Processing. Return a summary of public information concerning the purse (see below).

Field	Len.	Value	Meaning
CLA	1	'00'	ISO 7816 command without secure messaging
INS	1	'A4'	SELECT
P1	1	'04'	Select by name
P2	1	'00'	Only occurrence
Lc	1	5–16	Length of AID
Data	5–16	bytes	AID of the Demoney instance to be selected
Le	1	11	Maximum length of response data

Table 13: SELECT Command Message

Response Message. The format of the response message is given in Table 14. This command may return a general error condition as listed in Table 8.

Len.	Value	Format
1	Administrative status	see Table 6
10	Operational status (1)	see Table 7

(1) This item is present only if the application is already personalized.

Table 14: SELECT Response Message

5.8 INITIALIZE UPDATE

The INITIALIZE UPDATE command initiates a mutual authentication process. It is a specialization of the VOP 2.0.1' command with the same name[6].

Processing this command computes a cryptogram from a challenge sent by the terminal, and returns this cryptogram to the terminal for authentication, together with its own challenge. The command is to be immediately followed by an EXTERNAL AUTHENTICATE.

Note that we do not specify here the INITIALIZE UPDATE commands that may be sent to the card manager. We only specify the commands that are sent to the application or, possibly, to its security domain.

Command Message. The format of the INITIALIZE UPDATE command message is given in Table 15. The parameter P1 may have three different values in the range '01'–'7F' designating the debit, credit or administration key sets. These values are not specified of this document. P1 also determines the corresponding access level.

[6]Note that there is an inconsistency between OP 2.0 and OP 2.1.

- In OP 2.0, P1 is a key set version and P2 is a key index. It is P1 that determines a triplet of keys (K_{enc}, K_{mac}, K_{kek}) and P2 that determine which one of these three keys is to be used.
- In OP 2.1, P1 is a key version number and P2 is a key identifier. It seems that it is P2 that determines a key and P1 that tells which version of this key is to be used. The correspondence between a key identifier and a key set is undefined. For compatibility reasons, it is likely that the key set associated to a key identifier i consists of the keys identified by i, $i + 1$ and $i + 2$.

Field	Len.	Value	Meaning
CLA	1	’80’	OP command without secure messaging
INS	1	’50’	INITIALIZE UPDATE
P1	1	*var.*	Debit, credit or administration key set
P2	1	’01’	Key index 1
Lc	1	8	Length of data field
Data	8	binary	Host challenge
Le	1	28	Length of response data

Table 15: INITIALIZE UPDATE Command Message

Command Processing. Processing is as described in Section A.2.3:

- Close the current secure channel (if any) and reset the access level to *PUBLIC*.
- DEMONEY-STAND-ALONE and DEMONEY-OP-2.0: If the application is not personalized and P1 does not represent the administrative key set, return error “Incorrect parameters P1-P2”.
- DEMONEY-OP-2.1 only: If the application is not personalized, return error “Conditions of use not satisfied”.
- Generate an 8-bit challenge (the card random challenge),
- Generate a session key set from the card and host challenges using the static key set specified by P1,
- Generate a cryptogram using a combination of both challenges and the S-ENC session key.
- Temporarily record the access level corresponding to the key set indicated by P1 as well as the fact that the immediately preceding command was a successful INITIALIZE UPDATE.
- Return the card challenge, the cryptogram and additional key data (see below).

Response Message. The format of the response message is given in Table 16.

The key diversification data is not specified in this document. (It must be constant though.) It is not used by the INITIALIZE UPDATE command that are sent to the application. As for the command INITIALIZE UPDATE that are sent to the security domain of the application, it is as defined in a specific implementation of OP. (This information is specified for instance in VOP, not in OP.)

The key information data depends on the specification variant. The first byte of the key information data is the value of parameter P1. The second byte is, for DEMONEY-OP-2.1, the value ’02’ (the secure channel protocol identifier), and for the other variants, the value of parameter P2.

This command may return a general error condition as listed in Table 8.

Len.	Value	Format
10	Key diversification data	unspecified
2	Key information data	binary
8	Application challenge	binary
8	Application cryptogram	binary

Table 16: INITIALIZE UPDATE Response Message

5.9 EXTERNAL AUTHENTICATE

The EXTERNAL AUTHENTICATE command completes a mutual authentication process; it opens a secure channel session. It is a specialization of the VOP 2.0.1' and OP 2.1 command with the same name — and it is similar to the EXTERNAL AUTHENTICATE command in ISO 7816.

Processing this command checks a cryptogram computed by the terminal based on a challenge send by the card as a response to a previous command INITIALIZE UPDATE. Upon success, a session key is generated, based on both host and card challenges, to be used for secure messaging.

Command Message. The format of command EXTERNAL AUTHENTICATE is given in Table 17. The value of parameter P1 indicates a security level C-MAC (and R-MAC except in DEMONEY-OP-2.0). This security level applies to all secure messaging commands following this EXTERNAL AUTHENTICATE command (it does not apply to this command) and within this secure channel session.

Field	Len.	Value	Meaning
CLA	1	'84'	OP command with secure messaging
INS	1	'82'	EXTERNAL AUTHENTICATE
P1	1	'01' or '11'	Security level: C-MAC (and possibly R-MAC)
P2	1	'00'	unused
Lc	1	8 + 8	Length of data field
Data	8	binary	Host cryptogram
	8	binary	C-MAC
Le	0	empty	Expected response length

- DEMONEY-OP-2.0: P1 = '01' (C-MAC)
- DEMONEY-STAND-ALONE and DEMONEY-OP-2.1: P1 = '11' (C-MAC and R-MAC)

Table 17: EXTERNAL AUTHENTICATE Command Message

Command Processing. Processing is as described in Section A.2.3:

- DEMONEY-OP-2.1 only: If the application is not personalized, return error "Conditions of use not satisfied".
- If the *immediately* preceding command (since the last application selection) was not a *successful* INITIALIZE UPDATE, return error "Condition of use not satisfied".
- Compute a cryptogram by encrypting a given combination of both host and card challenges using the S-ENC session key stored and generated by the preceding INITIALIZE UPDATE command.
- Compare this cryptogram with the host cryptogram. If they do not match, return error "Authentication of host cryptogram failed".
- Set the access level to the level indicated in the previous INITIALIZE UPDATE command.

Response Message. The data field of the response message is not present. This command may return a general error condition as listed in Table 8.

5.10 PUT KEY

The PUT KEY command defines the keys of a given key set. It is a specialization of the VOP 2.0.1' command with the same name. The different variants of Demoney have different use of this command.

- DEMONEY-STAND-ALONE: The command is sent to the application. It sets application-specific keys.
- DEMONEY-OP-2.0: Before personalization, the command is sent to the security domain to set the personalization key set. After personalization, the command is sent to the application itself, to set the debit, credit and administration key sets.
- DEMONEY-OP-2.1: The command is sent to the security domain of the application, to define all key sets.

When sending this command to the application, the access level must be *ADMIN*.

The keys version numbers that are used by command PUT KEY are the same as those as indicated for command INITIALIZE UPDATE. Their actual value is not specified in this document.

Command Message. The format of the PUT KEY command message is given in Table 18.

Field	Len.	Value	Meaning
CLA	1	'84'	VOP command with secure messaging
INS	1	'D8'	PUT KEY
P1	1	*var.*	Debit, credit or administration key set
P2	1	'81'	Key index 1
Lc	1	67 + 8	Length of data field
Data	1	P1	Key set
	22	see Table 19	S-ENC key (key index 1)
	22	see Table 19	S-MAC key (key index 2)
	22	see Table 19	DEK (key index 3)
	8	binary	C-MAC
Le	1	10	Length of response data

Table 18: PUT KEY Command Message

Len.	Value	Format
1	Key type (DES)	'81'
1	Length of key data	16
16	Key data	binary
1	Length of key check value	3
3	Key check value	binary

Table 19: Key Data Field

Command Processing.

- DEMONEY-OP-2.1 only: Return error "Instruction code not supported or invalid".
- If P1 does not correspond to the debit, credit or administration key set, return error "Incorrect parameters P1-P2".

- If the access level is not *ADMIN*, return error "Security status not satisfied".
- Atomically: for each key data field,
  - Parse the key data field. If it is incorrect, return error "Incorrect parameters in the data field".
  - Decrypt the key data part of the key data field using the data encryption key of the current secure channel, i.e., the DEK of the administration key set.
  - Check the validity of the decrypted key data: the key check value consists of the 3 right-most (high order) bytes of the encryption of 8 binary zeros using this key. If a check value does not match, return error "Security status not satisfied".
  - Store the key in the key set designated by P1.

Response Message. The format of the response message is given in Table 20. It is a specialization of the VOP 2.0.1' format. This command may return a general error status as specified in Table 8.

Len.	Value	Meaning
1	P1	Updated key set number
3	binary	Key check value of the S-ENC key (key index 1)
3	binary	Key check value of the S-MAC key (key index 2)
3	binary	Key check value of the DEK (key index 3)

Table 20: PUT KEY Response Message

5.11 PIN CHANGE / UNBLOCK

The PIN CHANGE / UNBLOCK command is used to change the value of the PIN or to unblock it. It is a specialization of the VOP 2.0.1' command with the same name. The different variants of Demoney have different use of this command.

- DEMONEY-STAND-ALONE: The command is sent to the application. It sets or unblocks the application-specific PIN. Format and command processing are described below.
- DEMONEY-OP-2.0: The command is sent to the card manager. It sets or unblocks the global PIN. Format and command processing is described in VOP 2.0.1'.
- DEMONEY-OP-2.1: The command is sent to the card manager. It sets or unblocks the global PIN. There currently is no refinement of OP 2.1 defining this command. Format and command processing of VOP 2.0.1' is assumed.

The access level must be at least *ADMIN*.

Command Message. The format of the PIN CHANGE / UNBLOCK command message is given in Table 21. Parameter P2 indicates if the PIN is changed or unblocked.

- If P2 = 0, the PIN is to be unblocked. In this case, Lc = 0 and no PIN value is present within the command message.
- If P2 is in the range 3–15, the command is to set a new PIN. In this case, Lc = 16 and the command data field contains the new PIN value (encrypted with the DEK of the secure channel) and a MAC. The value P2 represents the retry limit of the PIN.

The PIN format is described in Table 22. It consists of 8 bytes, divided into 16 nibbles.

Field	Len.	Value	Meaning
CLA	1	’84’	OP command with secure messaging
INS	1	’24’	PIN CHANGE / UNBLOCK
P1	1	’00’	unused
P2	1	*var.*	Change / unblock option
Lc	1	0 or (8 + 8)	Length of data field
Data	8	see Table 22	Encrypted PIN
	8	binary	C-MAC
Le	0	empty	Expected response length

Table 21: PIN CHANGE / UNBLOCK Command Message

C L	P P	P P	P / F	P / F	P / F	P / F	F F

- C is the control character ’2’.
- L is the PIN length (number of Ps), in the range 4–12.
- P is a digit, in the range ’0’-’9’.
- F is a filler, with value ’F’.
- P / F means: either P P, or P F, or F F.

Table 22: PIN Format (Nibbles After Decryption)

Command Processing.

- DEMONEY-OP-2.0 and DEMONEY-OP-2.1 only: Return error “Instruction code not supported or invalid”.
- If the access level is not at least *ADMIN*, return error “Security status not satisfied”.
- If P2 $= 0$, unblock the PIN: if the PIN is currently undefined, return error “Conditions of use not satisfied” ; otherwise reset the number of times an incorrect PIN can be presented to its maximum value.
- If P2 $\neq 0$,
  - Decrypt the PIN value using the DEK of the secure channel in ECB mode (no padding or chaining required).
  - If the PIN syntax is incorrect, return error “Incorrect parameters in the data field”.
  - Atomically: set the new PIN value and retry limit, reset the number of times an incorrect PIN can be presented to its maximum value as defined at installation time, and mark the PIN as defined. Note that the actual value of P2 is ignored[7]

Response Message. The data field of the response message is not present. This command may return a general error condition as listed in Table 8.

[7]This restriction allows a Java Card implementation to use the `OwnerPin` class provided by the JC 2.1 API.

5.12 VERIFY PIN

The VERIFY PIN command performs the comparison in the card of the input PIN sent by the interface device with the reference user PIN stored in the application. It is a specialization of the ISO 7816 command VERIFY.

A PIN can be checked only a limited number of times. A counter keeps track of the remaining number of times an incorrect PIN can be presented. A PIN can be in three states:

- validated (and not blocked): a correct PIN has been presented (and thus the remaining number of times an incorrect PIN can be presented is not null).
- neither validated nor blocked: no correct PIN has been presented and the remaining number of times an incorrect PIN can be presented is not null.
- blocked (and not validated): no correct PIN has been presented and the remaining number of times an incorrect PIN can be presented is null.

It is not possible for the PIN to be both validated and blocked.

Command Message. The VERIFY PIN command message is coded according to Table 23. The PIN format is given in Table 22.

Field	Len.	Value	Meaning
CLA	1	’94’	Demoney command with secure messaging
INS	1	’20’	VERIFY instruction
P1	1	’00’	unused
P2	1	’00’	unused
Lc	1	8	Length of the PIN
Data	8	see Table 22	PIN
Le	0	empty	Expected response length

Table 23: VERIFY PIN Command Message

Command Processing.

- If the application is not personalized, return error “Conditions of use not satisfied”.
- If the user PIN is not defined, return error “Function not supported”.
- Check that the PIN is not blocked, i.e., that the remaining number of times an incorrect PIN can be presented is strictly positive; if not so, return error “Authentication method blocked”.
- Atomically,
  - Decrement the remaining number of times an incorrect PIN can be presented.
  - Compare the input PIN sent by the CAD to the user PIN.
  - Set the PIN validity flag accordingly.
  - In case of success, reset the number of times an incorrect PIN can be presented to its maximum value; in case of failure and if the number of times an incorrect PIN can be presented is null, the PIN is considered blocked.
- If the PIN validity flag is unset, return error “PIN verification failure”.

Response Message. The data field of the response message is not present. This command may return a general error condition as listed in Table 8.

5.13 INITIALIZE TRANSACTION

The INITIALIZE TRANSACTION command tests through a secure channel if a transaction (debit, credit from cash or credit from bank) is accepted by the application. The command is to be immediately followed by a COMPLETE TRANSACTION command to validate the operation. Secure messaging is as follows. (See also §2.2.)

- Mutual authentication between the application and the terminal is required: the access level must be at least *DEBIT*, *CREDIT*, or *CREDIT-IDENT* (see §4.3), depending on the type of the transaction.
- The command message is signed with the C-MAC session key of the corresponding secure channel.
- The response message is signed with the R-MAC session key of the secure channel. This signature is used by the POS as a certificate that the INITIALIZE TRANSACTION command was successful. Without a proper MACed response, the INITIALIZE TRANSACTION is to be considered unsuccessful for security reasons, even if the POS receives the success status word '90 00' (see §2.2).

In case of a credit from a bank account, the response data includes bank information so that the terminal knows what bank account is to be debited at the same time as the purse is credited.

Command Message. The INITIALIZE TRANSACTION command message is coded according to Table 24. Parameter P1 indicates the transaction type: debit ('00'), credit from cash ('01'), or credit from a bank account ('02').

Field	Len.	Value	Meaning
CLA	1	'94'	Demoney command with secure messaging
INS	1	'44'	INITIALIZE TRANSACTION
P1	1	'00'-'02'	Transaction type
P2	1	'00'	unused
Lc	1	4 + 8	Length of data field
Data	2	unspecified	Transaction currency
	2	positive signed integer	Transaction amount
	8	binary	C-MAC
Le	1	(4 or 20) + 8	Length of response data

Table 24: INITIALIZE TRANSACTION Command Message

Command Processing.

- If the application is not personalized, return error "Conditions of use not satisfied".
- If P1 indicates credit from bank account and either the bank information parameter or the PIN is undefined, return error "Function not supported".

- If the access level is not at least *DEBIT* for a debit transaction, *CREDIT* for a credit from cash, or *CREDIT-IDENT* for a credit from a bank account, return error “Security status not satisfied”.
- If the security level is not C-MAC and R-MAC, return error “Security status not satisfied”.
- If the purse transaction number is greater or equal to the maximum transaction number then return error “Maximum number of transactions reached”.
- If the transaction currency is different from the purse currency, return error “Currency error”.
- If the transaction amount is not strictly positive, return error “Invalid transaction amount”.
- If the transaction is a debit: if the debit amount is greater than the maximum transaction amount or if the current balance is lesser than the debit amount, return error “Debit amount too high”.
- If the transaction is a credit: if the balance incremented by the credit amount is greater than the maximum balance, return error “Credit amount too high”.
- Temporarily record the tentative transaction amount (until next command).
- Report that the transaction is possible. Include bank information in the response message if P1 indicates credit from bank account.

Response Message. The format of the response message is given in Table 25. The bank information is only present if the command indicates a credit from a bank account. If present, it is ciphered with the DEK key (in CBC mode). This command may return a general error status as specified in Table 8 or a specific error status as specified in Table 9.

Len.	Value	Format
4	Purse identifier	unspecified
16	Bank information (1)	unspecified
8	R-MAC	binary

(1) Only present (encrypted) if P1 indicates credit from a bank account.

Table 25: INITIALIZE TRANSACTION Response Message

5.14 COMPLETE TRANSACTION

The COMPLETE TRANSACTION command completes a transaction operation previously started by an INITIALIZE TRANSACTION command. As the INITIALIZE TRANSACTION command, it operates through a secure channel. Secure messaging is as follows. (See also §2.2.)

- This command must be immediately preceded by a successful INITIALIZE TRANSACTION command.
- Mutual authentication between the application and the terminal is required and the access level must be at least *DEBIT*, *CREDIT*, or *CREDIT-IDENT* (see §4.3), depending on the type of transaction. (In practice, it is enough to know that the immediately preceding command was a successful INITIALIZE TRANSACTION command.)
- The command message is signed with the C-MAC session key of the corresponding secure channel.
- The response message is signed with the R-MAC session key of the secure channel. This signature is used by the POS as a certificate that the COMPLETE TRANSACTION command was

successful. Still, even if the COMPLETE TRANSACTION does not succeed for some reason, the terminal can record a credit failure but can never assumes that the credit was certainly not performed (see §2.2). Similarly, without a proper MACed response, the debit is to be considered unsuccessful for security reasons, even if the POS receives the success status word '90 00'.

Command Message. The COMPLETE TRANSACTION command message is coded according to Table 26.

Field	Len.	Value	Meaning
CLA	1	'94'	Demoney command with secure messaging
INS	1	'46'	COMPLETE TRANSACTION
P1	1	'00'	unused
P2	1	'00'	unused
Lc	1	16 + 8	Length of data field
Data	16	see Table 4	Transaction context
	8	binary	C-MAC
Le	1	16 + 8	Length of response data

Table 26: COMPLETE TRANSACTION Command Message

Command Processing.

- If the application is not personalized, return error "Conditions of use not satisfied".
- If the *immediately* preceding command (since the last application selection) was not a *successful* INITIALIZE TRANSACTION, return error "Condition of use not satisfied".
  (A successful preceding INITIALIZE TRANSACTION command implies that the current access level and security level are appropriate, and that the limit on the transaction number is not reached, see §5.13; there is no need to check those conditions again, see §4.6.)
- Atomically:
  - Increment (if transaction is a credit) or decrement (if transaction is a debit) the balance by the transaction amount that was specified in the previous INITIALIZE TRANSACTION command.
  - Increment the purse transaction number.
  - Record the transaction in the cyclic log file as indicated in Table 3.

Response Message. The format of the response message is given in Table 27. This command may return a general error status as specified in Table 8 or a specific error status as specified in Table 9.

5.15 GET DATA

The GET DATA command is used for the retrieval of data stored in the application. More precisely, it can return the value of configuration parameters and state variables. It is a specialization of the Open Platform command with the same name. There are three kinds of information:

- public information, such as the current balance, that is available at any access level,

Len.	Value	Format
4	Purse identifier	unspecified
2	Purse transaction number	unsigned integer
10	Operational status	See Table 7
8	R-MAC	binary

Table 27: COMPLETE TRANSACTION Response Message

- inaccessible information, such as the keys and the PIN code, that are unavailable at any access level.

The command GET DATA can be used at any time after installation, including before personalization. In particular, the administrative status of the application (see §3.4.1) can thus be read at any time.

Command Message. The format of the GET DATA command message is given in Table 28. P2 indicates the tag of the data to retrieve, as specified in Tables 1 and 2.

Field	Len.	Value	Meaning
CLA	1	'80'	OP command without secure messaging
INS	1	'CA'	GET DATA
P1	1	'02'	TLV tag in P2
P2	1	*var.*	Tag of the data to retrieve
Lc	0	—	empty
Data	0	—	empty
Le	1	*var.*	Maximum length of response data

Table 28: GET DATA Command Message

Command Processing.

- If the tag is not a valid tag as defined in Tables 1 and 2, return error "Incorrect parameters P1-P2".
- Return value corresponding to the tag in the response message (see below).

Response Message. The format of the response message is given in Table 29; it is the data in LV form, formatted as indicated in Tables 1 and 2. If an optional parameter is requested and this parameter is unset, the value length is null and the value field is empty. This command may return a general error status as specified in Table 8.

Len.	Value	Format
1	Length of value	unsigned integer
var.	Value corresponding to tag	*var.*

Table 29: GET DATA Response Message

5.16 PUT DATA

The PUT DATA command is used for storing data into the application as well as to unblock the PIN. More precisely, it can set the value of configuration parameters. It is a specialization of the VOP 2.0.1' command with the same name. It can only be performed at access level *ADMIN*.

Command Message. The format of the PUT DATA command message is given in Table 30. P2 indicates the tag of the parameter to update, as specified in Table 1; Lc is the length of the parameter; and the data field contains the new value of the parameter. In fact, P2-Lc-Data (without the MAC) represents the parameter update in TLV form.

Field	Len.	Value	Meaning
CLA	1	'84'	OP command with secure messaging
INS	1	'DA'	PUT DATA
P1	1	'02'	TLV tag in P2
P2	1	*var.*	Tag of the data to update
Lc	1	*var.*	Length of data field
Data	*var.*	*var.*	New data value
	8	binary	C-MAC
Le	0	—	empty

Table 30: PUT DATA Command Message

Command Processing.

- If the application is not personalized, return error "Conditions of use not satisfied".
- If the access level is not *ADMIN*, return "security status not satisfied".
- If the tag is unknown (i.e., not in Tables 1 or 2), return error "Incorrect parameters P1-P2".
- If the tag is known but corresponds to a state variable (Table 2) rather than to a configuration parameter (Table 1), return error "Function not supported".
- If the tag corresponds to the purse identifier, return error "Function not supported".
- If the tag corresponds to the purse currency and the current balance is not null, return error "Function not supported".
- Atomically:
  - If the length is not in the expected range for the parameter indicated by the tag (including possible or impossible value zero), return error "Incorrect parameters in the data field".
  - If the length is null, consider now the value as undefined. (Mandatory parameters cannot be undefined because of the previous verification concerning lengths.)
  - If the length is not null, consider now the value as defined and store it in persistent memory.
  - If the final configuration of persistent data is inconsistent (see §3.2.4), return error "Incorrect parameters in the data field".

Note that, because the above operation is atomic, the state of the application is not actually updated if the data configuration is inconsistent.

Response Message. The data field of the response message is not present. This command may return a general condition error as listed in Table 8.

5.17 READ RECORD

The READ RECORD command is used for the retrieval of records in the transaction log file of the application (see §3.2.3). It is a specialization of the ISO 7816 command with the same name.

Command Message. The format of the READ RECORD command message is given in Table 31. P1 indicates the number of the record to read (see below).

Field	Len.	Value	Meaning
CLA	1	’00’	ISO 7816 command without secure messaging
INS	1	’B2’	READ RECORD
P1	1	1–50	Record number
P2	1	’0C’	Read record number P1 of EF number 1
Lc	0	—	empty
Data	0	—	empty
Le	1	18	Length of response data

Table 31: READ RECORD Command Message

Command Processing.

- If the application is not personalized, return error “Conditions of use not satisfied”.
- If there is no record numbered P1 (either because P1 is greater than the maximum number of records set at installation time or because there was not that many transactions yet), return error “Record not found”.
- Return the log file record corresponding to the number P1 (see Section 3.2.3).

Response Message. The format of the response message is given in Table 3. This command may return a general error condition as listed in Table 8.

A Secure Channels

This appendix describes *secure channels* based on authentication, message signature and encryption, to protect communication from identity usurpation, forging, replay and observation of private data. These secure channels are based on those described in the Open Platform architecture.

A.1 Secure Channel Basics

This section provides the basics of secure messaging via secure channels. The actual model defined in OP is described in Section A.2.

A.1.1 Authentication Basics

Authentication prevents usurpation of identity using transaction forging and replay. It is a three step process:

1. A party A that requires another party B to be authenticated issues a *challenge* (e.g., an incremented counter, a random number) and sends it to B.
2. Party B encrypts the challenge using a secret key K that is shared by both A and B, and returns this *cryptogram* to A.
3. Party A perform the same computation using its own version of the key K and compares the resulting cryptogram with the one that was sent by B. If both cryptograms match, then A and B share the same key K and A can consider B as *authenticated*.

The fact that the challenge is always different guarantees that the authentication transaction cannot be replayed, hence that the identity of B cannot be usurped.

To prevent replays of the following messages in another session, messages can be signed or encrypted using a *session key* derived from the *static key* K and the challenge, now known both by A and B.

Mutual authentication require both A and B to issue challenges. A session key can then be based on K and on a combination of both challenges. If more that one session key is needed, different session keys may be generated from different static keys and different combination of the challenges.

A.1.2 Message Signature Basics

To prevent transaction forging, messages from a party A to a party B have to be signed to guarantee both message integrity and sender authentication. The signature is based on a secret key that is shared by both A and B.

1. A message to be sent by A to B is hashed using the secret key and the resulting *hash* is appended to the message before being sent. This hash, also called MAC (Message Authentication Code) in Open Platform, is the *signature* of the message.
2. When receiving the message, including the signature, B computes a hash of the message using the same algorithm and data as A. B then compares this hash to the signature present in the message. If the message was altered during transmission, then the hash and the signature differ. If the hash and the signature are equal, B can safely assume that the message was sent by A and that it was not forged.

The key used to sign messages can be permanently stored in A and B, or can be a session key constructed during a preliminary authentication phase.

A.1.3 Protection against Replay within the Session

Even if a secure channel is open and session keys are used, the same message could be replayed in the same session, e.g., to reiterate a credit.

As a protection against replay, a message from A to B may include in its body some specific data sent by B to A in a preceding message. These data have to be unique (or statistically unique) in a session; it can be a counter (or random number). The protection for B consists in checking that the data earlier sent to A and the data newly sent by A are equal. The signature guarantees that only A can have built the message and that it is unaltered; the uniqueness of the data guarantees that the message cannot be replayed.

An alternate protection against replay within the session consists in using the previous signature to generate the next one. Hence, the signature of two identical messages (statistically) differ. Moreover, chaining signatures also captures the sequencing of commands: if the commands are presented in the wrong order, the signatures do not match. This anti-replay mechanism is used in the signature model of Open Platform. This is also the one we use in Demoney.

A.1.4 Message Encryption Basics

To prevent a third party to observe the contents of messages exchanged between A and B, these messages can be encrypted. This is generally done using a session key.

Alternatively, because encryption (and decryption) is a heavy process, it can be limited to only part of the message data: the sent data then consists of public data in plaintext and secret data in crypted form. If the data to encrypt can have any value (or statistically almost any value), which generally is the case for key data, it is not necessary to use a session key; a static key is enough.

Some data may have their own encryption procedure. For example, key data may include some kind of checksum. (This is the case in Open Platform.)

In some cases, a partly encrypted message can in addition be encrypted as a whole. The two corresponding keys do not have to be the same. (It is not considered a good practice to cipher an already ciphered text with the same key.)

In any case, messages are not only encrypted but also signed, to guarantee origin and integrity. The message encryption key(s) and the message signature key are not necessarily equal. As a matter of fact, they are different in OP.

A.2 Secure Channels *à la* Open Platform

This section describes the secure channels defined in Open Platform. The actual support of an OP system is not required though: this is just the specification of protocols and algorithms, that can be implemented on a card without OP. As a matter of fact, these secure channels are used for both variants DEMONEY-STAND-ALONE and DEMONEY-OP-2.0.

All keys used for secure messaging in Open Platform are 2-key triple DES keys; they are 16 bytes long. This includes static keys as well as session keys.

A.2.1 Security Levels

Secure channels in Open Platform have a *security level* that expresses whether the command messages and/or response messages should be signed and possibly encrypted. In the case of Demoney, we use three security levels

- no security (i.e., no secure channel),

- C-MAC: signed command messages,
- C-MAC and R-MAC: signed command and response messages.

In addition to the security level, all or part of the data carried in a message can be encrypted (e.g., keys).

Security level is an attribute of the secure channel that is defined when the secure channel is open. In OP, it cannot be later modified; a new secure channel has to be open if a new security level is required.

A.2.2 Secure Channel Keys and Key Sets

For security reasons (see §2.2), a secure channel relies both on *static keys*, that live permanently[8] on both the card and on the host, and *session keys*, that only live between the opening of a secure channel and its closing. Besides, secure channels in OP use four kinds of keys, corresponding to different security issues.

- The *encryption session key* is used both for mutual authentication and (whole) message encryption. As matter of fact, VOP calls this key the *Encryption/Authentication session key*, also noted $K_{\text{enc/auth}}$.
- The *C-MAC session key* is used for command message signature. In OP 2.0, only the command message can be signed; they is just called the *MAC session key*, also noted $K_{\text{enc/auth}}$.
- The *R-MAC session key* is used for response message signature. This key only exists in OP 2.1 with secure channel protocol SCP02.
- The static *data encryption key* (DEK) is used to encrypt sensitive data. This is the OP 2.1 terminology. In OP 2.0, this key is called the *key encryption key* (KEK, also noted K_{KEK}) rather than DEK, as it was originally intended to communicate encrypted secret keys.
  Note that it is not a session key but a static key. In practice, either the whole message in encrypted, using the encryption session key, or only part of it are encrypted, using the DEK. It rarely is the case that a message not only contains encrypted data, but also is encrypted as a whole.

The session keys are generated from session key derivation data and from static keys. The *session key derivation data* are made of a combination of host and card challenges in OP 2.0; they are made of a card sequence counter in OP 2.1.

The static keys are grouped into a *key set* (terminology stressed in OP 2.0 and forgotten in OP 2.1) that is designated when opening a secure channel. A key set consists of:

- a *secure channel encryption key* (S-ENC key, also noted K_{ENC}), used to generate the encryption session key,
- a *secure channel MAC key* (S-MAC key, also noted K_{MAC}), used to generate the C-MAC and R-MAC session keys,
- a *secure channel data encryption key* (DEK, or KEK in OP 2.0, also noted K_{KEK}), used as a static key to encrypt sensitive data.

A key set may be reduced to a single key, called *secure channel base key*. In this case, this key is used at the same time as an S-ENC key, an S-MAC key and a DEK.

The encryption session key is always used, even if message encryption is not needed, because it is also required for mutual authentication. The C-MAC session key is only used if the secure channel

[8] In the sense that they are persistent data; they can be redefined.

requires command message signature. Similarly, the R-MAC session key is only used if the secure channel requires response message signature. The DEK is only used if the application requires data encryption.

A.2.3 Mutual Authentication

Mutual authentication is the authentication of the card by the host (i.e., the terminal, the CAD) and authentication of the terminal by the card. Mutual authentication requires two APDU commands.

1. An INITIALIZE UPDATE command[9] allows the host to authenticate the card.
2. An EXTERNAL AUTHENTICATE command allows the card to authenticate the host.

Mutual authentication is based on a secret static encryption key K_{ENC} only known to the card and the host.

The following paragraphs detail mutual authentication in variants DEMONEY-STAND-ALONE and DEMONEY-OP-2.0. The protocol and algorithms are exactly those defined in OP 2.0 (as well as secure channel protocol SCP01 of OP 2.1, see §2.4). DEMONEY-OP-2.1 relies on the secure channel protocol SCP02 of OP 2.1, that uses a different algorithm to compute session keys (not described here).

Card Authentication. The first authentication stage is as follows.

- The host generates a random challenge H and sends it to the card via an INITIALIZE UPDATE command. The challenge H is 8-byte long.
- When the card receives the H, it generates its own random challenge C. The challenge C also is 8-byte long.
- The card then computes the encryption/authentication session key $K_{\text{enc/auth}}$. For this, both H and C are split into H_1, and H_2, C_1 and C_2, representing respectively the first and second 4-byte halves of each 8-byte challenge. $K_{\text{enc/auth}}$ is then computed by ciphering, in ECB mode, with key K_{ENC}, the block C_2-H_1-C_1-H_2.
  The MAC session key $K_{\text{enc/auth}}$ is generated exactly as the encryption session key, using K_{MAC} instead of K_{ENC}. It is not used for authentication though; it is only used later on, after authentication, for verifying the signature of command messages.
- The card then computes a cryptogram G by signing the block H-C. The signing method consists of ciphering H-C, padded with a further 8-byte '80 00 00 00 00 00 00 00', in CBC mode, with the session key $K_{\text{enc/auth}}$, and with an initial chaining vector (ICV) equal to 8 binary zeros. The signature is the last eight bytes of the ciphered block.
- The card returns to the host both its random challenge C and the cryptogram G it has computed. Note that, at this point, both the card and the host know both challenges, H and C.
- Finally the host computes both an encryption/authentication session key and a cryptogram using exactly the same procedure as above. If the cryptogram sent by the card is equal to the one computed by the host, then the card is considered authenticated: the host has the proof that the card possesses the same encryption/authentication session key, and therefore the same encryption key K_{ENC}.
  The host also generates at this point its own copy of the MAC session key $K_{\text{enc/auth}}$. It is not used for authentication; it is only used later on, after authentication, for signing command messages.

[9] INITIALIZE UPDATE is the Open Platform name of the command. The ISO 7816 equivalent command is INITIALIZE UPDATE.

Host Authentication. The second authentication stage is as follows.

- The host computes its own cryptogram G' by signing the block C-H with the session key $K_{\text{enc/auth}}$. Note that the challenges are permuted. The signature consists in ciphering C-H, padded with a further 8-byte '80 00 00 00 00 00 00 00', in CBC mode, with the session key $K_{\text{enc/auth}}$, and with an ICV equal to 8 binary zeros. The signature is the last eight bytes of the ciphered block.
- The host send its cryptogram G' to the card via an EXTERNAL AUTHENTICATE command.
- The card then computes its own cryptogram based on the same reversed block C-H and using its own session key $K_{\text{enc/auth}}$. If the cryptogram computed by the card is equal to the cryptogram computed by the host, then the host is considered authenticated: the card has the proof that the host possesses the same session key $K_{\text{enc/auth}}$, and therefore the same encryption key K_{ENC}.

At this point, both the card and the host operate with the same session key $K_{\text{enc/auth}}$ for message encryption. (No authentication is further required.) They also share the same $K_{\text{enc/auth}}$. (More precisely, they share the same $K_{\text{enc/auth}}$ only if they share the same K_{MAC}. If they do not, signature verification will always fail. The application will be mostly inoperative but security will not be broken.)

A.2.4 Signing Command Messages

A certified APDU is an APDU that contains a digital signature. In Demoney, as in OP, the signature (MAC) is 8 bytes long. It is computed as follows.

- The length of the command message ("Lc" byte) is incremented by 8, i.e., the MAC length.
- The class code ("CLA" byte) is modified to indicate that the APDU includes secure messaging: the third bit is set.
- The APDU, striped from the response length byte ("Le" byte), if any, is always padded with '80', and then possibly with zeros until its length is a multiple of eight.
- The MAC is computed by ciphering the APDU in CBC mode with the C-MAC session key $K_{\text{enc/auth}}$. The ICV depends on the command:
  - For the EXTERNAL AUTHENTICATE command, the ICV is equal to 8 binary zeros.
  - For any command following EXTERNAL AUTHENTICATE, the ICV is the C-MAC successfully verified for the previous signed command message received by the card.

  The MAC is the last eight bytes of the ciphered block.
- The padding is removed from the APDU.
- The MAC is included into the APDU Data field.
- The resulting APDU is sent to the card.

To verify the signature, the card operates as follows.

- The host MAC is removed from the APDU Data field.
- A card MAC is computed from the striped APDU exactly as described above for the host.

If both MACs match, then the signature is correct and the card can consider not only that the APDU was sent by the host but also that it was not altered during communication.

A.2.5 Signing Response Messages

The response signature is computed on a data block made of the following successive components:

- the unwrapped APDU command message, i.e., the header with class code having the third bit unset and the command data field without C-MAC,
- an unsigned byte indicating the length of the response data (without status words),
- the response data,
- the two bytes of the status word,
- a padding consisting of the byte '80', and then possibly with binary zeros until its response length is a multiple of eight.

The signature consists in ciphering this data block, in CBC mode, with the R-MAC session key, and with an ICV equal to 8 binary zeros. The signature is the last eight bytes of the ciphered block. The sign message that is returned to the terminal consists of:

- the response data,
- the response MAC,
- the status word.

The R-MAC session key differs according to the Demoney variant:

- DEMONEY-STAND-ALONE: The C-MAC session key $K_{\text{enc/auth}}$ is also used as the R-MAC session key.
- DEMONEY-OP-2.0: OP 2.0 does not define the signature of response messages in a secure channel. More than that, session keys are not made available to sign (or encrypt) a message. As for DEMONEY-STAND-ALONE, we also use the C-MAC session key $K_{\text{enc/auth}}$ as the R-MAC session key for this variant.
- DEMONEY-OP-2.1: OP 2.1 (with SCP02) specifies how response messages should be signed (this is the algorithm defined above) and makes available the R-MAC session key to do so. DEMONEY-OP-2.1 uses this R-MAC session key (not defined here), which is different from the C-MAC session key.

Note that the need for signed response messages has had a major impact on the design of Demoney. In particular, as DEMONEY-OP-2.0 cannot solely rely on keys managed by Open Platform, it has to manage its own key sets to be able to sign responses. The need of response messages also explains why the security domain of DEMONEY-OP-2.1 must support the secure channel protocol SCP02.

B Java Card Implementation

This appendix lists the APIs that Java Card implementations of the different Demoney variant must or may use. It also defines how JC implementations of Demoney have to notify a purchase agent.

B.1 Java Card 2.1.1 API

The Java Card 2.1.1 API includes the following features (among others):

- Key management: definition, encryption, decryption,
- Signature management: signature generation and verification,
- Random number generation,
- PIN management: definition, verification,
- Session data: transient arrays,
- Atomicity: starting, committing and aborting transactions.

It is recommended for a Java Card implementation of DEMONEY-STAND-ALONE to use all the above features. An implementation of DEMONEY-OP-2.0 does not need the PIN support as it is handled by OP 2.0. An implementation of DEMONEY-OP-2.1 only needs support for atomicity as all other issues are handled by OP 2.1.

B.2 Open Platform 2.0 API

The Open Platform 2.0 Java Card API provides the following features (among others):

- Full processing of commands INITIALIZE UPDATE and EXTERNAL AUTHENTICATE,
- APDU unwrapping: signature verification and, possibly, message decryption,
- Key decryption and verification,
- Global PIN management.

All these features have to be used to implement DEMONEY-OP-2.0 because this specification variant relies on resources that are managed by OP rather than by the application itself: secure channel based on the personalization key set, global PIN.

B.3 Open Platform 2.1 API

The Open Platform 2.1 Java Card API provides the following features (among others):

- Full processing of commands INITIALIZE UPDATE and EXTERNAL AUTHENTICATE,
- Command message unwrapping: signature verification and, possibly, message decryption,
- Response message wrapping: signature generation,
- Message data decryption and encryption,
- Global PIN management.

All these features have to be used to implement DEMONEY-OP-2.1 because this specification variant relies on resources that are managed by OP rather than by the application itself: keys of all secure channels, global PIN.

B.4 Purchase Agent Notification

Demoney also has a mechanism to notify purchases to another application present on the card, such as a loyalty application or a budget manager. In a Java Card implementation, this mechanism is based on a shareable interface.

When a debit is performed in Demoney, if the AID of a purchase agent is defined, then the following operations are performed:

- A shareable object associated to this purchase agent (via its AID) is requested to the system. The byte parameter to the request method `getAppletShareableInterfaceObject` must be zero.
- This shareable object is cast to the following interface:

```
package fr.trustedlogic.demo.purchaseAgent;

interface PurchaseAgent
{
  public void notifyPurchase(byte[] buffer, short offset);
}
```

- The `notifyPurchase` method is called with a buffer[10] loaded with the information described in Table 32.

Len.	Value	Format
2	Transaction currency	unspecified
2	Purchase amount	positive signed integer
16	Transaction context	see Table 4

Table 32: Purchase Agent Information

Because a given card can run Demoney, or the purchase agent, or both, the `PurchaseAgent` interface cannot be in Demoney's package nor in the actual purchase agent package; it has to be in a separate package. This package is to be loaded before Demoney or the purchase agent is loaded onto the card.

Other payment applications may also communicate to a purchase agent via the same interface.

[10] In fact, because of the Java Card firewall, only the APDU buffer can be read by another applet. The argument buffer of the method `notifyPurchase` can thus only be the APDU buffer.

References

[BCM+00] P. Bieber, J. Cazin, A. El Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G.Zanon. The PACAP prototype: a tool for detecting Java Card illegal flow, September 2000. Java Card Forum 2000, Cannes, France.

[BMGL01] E. Bretagne, A. El Marouani, P. Girard, and J.-L. Lanet. PACAP purse and loyalty specification, January 2001. Gemplus, v0.4.

[Gem] Gemplus. Applet benchmark kit. http://www.gemplus.com/smart/r_d/publications/case-study/index.html.

[Glo01] Global Platform. Open Platform card specification v2.1, June 2001.

[Sun98] Sun Microsystems. Java Card 2.0 Reference Implementation User's Guide, February 1998.

[Sun01a] Sun Microsystems. Java Card 2.1.2 Development Kit, 5 April 2001. See http://java.sun.com/products/javacard/dev_kit.html.

[Sun01b] Sun Microsystems. JavaPurse, March 2001. Provided as sample applet in [Sun01a].

[Sun01c] Sun Microsystems. Wallet, March 2001. Provided as sample applet in [Sun01a].

[VIS99a] VISA. Open Platform card specification v2.0, 19 April 1999.

[VIS99b] VISA. Open Platform card specification v2.0.1, 31 December 1999.

[VIS00a] VISA. Open Platform card specification v2.0.1', 7 April 2000.

[VIS00b] VISA. Visa Open Platform v2.0.1', visa card implementation guide, configuration 1 – compact, 30 September 2000.

[Wie00] Virginie Wiels. PACAP project, 2000. See http://www.cert.fr/francais/deri/wiels/Pacap/.