Point Of Sale/Service and MER Interaction

SAP On Device Charging Development Guide

Sybase Mobiliser Platform 5.1 SP03

Document ID:

Last Revised: July 2013

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Table Of Contents

Welcome to SAP On Device Charging

SAP On Device Charging is built upon a powerful, flexible pricing, and charging solution. Its intuitive GUI drives a powerful set of tools to profitably price and rate transaction directly on end user's device. SAP On Device Charging's modular design includes a framework for fast and low-cost integration to quickly deploy solutions that enable you to get a rapid return on your investment.

Purpose of this guide

This development guide describes how to interact device from the Point of Service with the ODC MER (Mobile Embedded Rater) that is installed on a customer's. It proposes:

- A general overview about the ODC architecture.
- The POS APIs provided by the MER.
- A brief description about ISO7816
- A description of APDUs used by a POS to communicate with a MER.
- A description of the security protocol between the POS and the MER.
- The java POS library (com.sap.odc.mer.api.pos) proposed by ODC for a high level integration.
- The SAP Cola full code that can be used as an example / tutorial.
- An extension to the POS APIs: the READ API.
- A description of APDU's proposed by the READ API.
- The java READ library (com.sap.odc.mer.api.read) proposed by ODC for a high level integration.

Readers

SAP On Device Charging's development guide is intended for power developers who create POS application and manage the POS for the service provider. They should have knowledge of:

- NFC, Javacard and ISO7816
- Encryption libraries
- ODC Charging concepts (Chargeable Items, Charging and Pricing plan)

Guide version

Document version: 0.1 - July 2013

SAP On Device Charging overview

About SAP On Device Charging

SAP On Device Charging is a modular solution for pricing and charging processes performed by the mobile handset. You can:

- Use the panel of tools to streamline the way you create price plans and charging plans
- Build up complex offers without writing a line of code
- Use the intuitive GUI which enables "visual programming" to
  - ✓ visualize subscriptions
  - ✓ customize your price algorithms
- Subscribe to offers defined by the merchants
- Synchronize the user subscriptions (price / charge plans, counters and Prepaid Accounts) with remote devices for executing business logic locally on the device in a secure way.

To help you get the most out of SAP On Device Charging®, SAP AG has set up a Help Portal which features a regularly updated knowledge base, the latest release notes, and expert advice from our Support team.

Setting the context

ODC is the On-device version of the centralized SAP Convergent Charging (SAP CC) software.
ODC supports the SAP CC main features

- Manage Stored Value Card also called offline SVA on the device
- Rate/charge any kind of service
- Implement some flexible business logics
- Cascade and compose price plans for multi-partners business models

Where can it be implemented?

- On Javacard Secure Element used by devices,
- Ex devices : *plastic cards, mobile phones (SIM), on-board-units, set-top-boxes, tablets, smart meters, ...*
- Directly on smart phones (unsecure), to estimate price and control the service costs

What does it do?

- Allows millions of devices to perform in a secure way the following functions locally on the device itself:
  - Business logic execution
  - Rating & charging
  - Stored value accounts management
- Works in both off-line and on-line modes

What you can do in SAP On Device Charging

The main features of SAP On Device Charging are the following:

Catalogs
To help you organize your work more efficiently, you create catalogs for storing all the objects that are needed for the service provider.

Chargeable Item packages
You need to define Chargeable Item packages that represent services which can be rated according to their usage. With NFC capabilities the Chargeable Items will be transmitted by the point of service through the NFC reader.

Charges
You create charges that define the price plan and the charging plan to be applied to the usage of a service.

Offers
You create offers which represent the way that you sell services.

Pricing macros
You can create pricing macros that are small reusable modules for applying calculation formulas within the charges.

Translation tables
You can create translation tables used within charges and refill logic to assign specific output values to input values.

Subscriber Accounts
Before creating subscriptions, you must create one or more subscriber accounts representing customers in SAP On Device Charging. If you work with *Sybase Money Mobiliser*, the corresponding subscriber accounts are automatically created in SAP On Device Charging. The access to existing subscriber accounts is available in read-only mode through the SAP On Device Charging Core Tool.

Subscriptions
You can allow customers to subscribe to one or more offers proposed by service providers and manage all the conditions relating to subscriptions. If you work with *Sybase Money Mobiliser*, the subscriptions will be automatically created by *Sybase Money Mobiliser*. The access to existing subscriptions is available in read-only mode through the SAP On Device Charging Core Tool.

Accesses
At subscription or provider contract level, the user access specifies the identifier under which a customer consumes a service.

User profiles
You can assign a login, a password, and one or more roles to each SAP On Device Charging's user. But if you work with *Money Mobiliser* the ODC user profile creation is completely handled by *Sybase Money Mobiliser*.

Audits
SAP On Device Charging can save user operations to the database for audit purposes. You can create filters to search for audited user operations in your database.

Tools
You are provided with specific tools to work with SAP On Device Charging.

Synchronization
The user can synchronize/push the subscriber accounts and subscriptions on the customer's device (mobile handset in most of time) by using OTA infrastructure. If you work with *Sybase Money Mobiliser*, the synchronization is managed by Money Mobiliser but can be triggered by the end customer from his *Sybase Mobiliser Smartphone* application running on his mobile.

Architecture overview

Actors / Roles

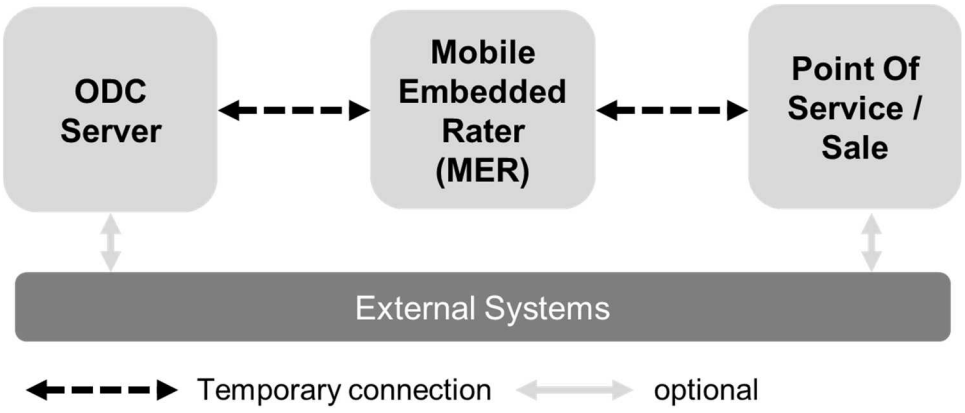
The following roles/actors can be defined with SAP-ODC:

- **Vendor / Merchant / Service Provider**
  - Party who sells a good or a service (tangible or intangible)
  - Party who creates offers and Prepaid Accounts but who doesn't operate a SAP ODC platform.
  - A vendor needs a SAP-ODC operator to publish his offers.
  - A vendor can also delegate the offer's implementation to the operator.
  - A vendor can propose some Point of Sale / Service that can interact with the Customer
  - A vendor can use some third POS which are proposed by the operator

- **Operator**
  - Party who manages a SAP-ODC platform.
  - An operator implements the vendors' offers for the customers through his platform.
  - An operator can create his own offers and provide a service; in this case he plays a vendor role too.
  - An operator can also manage and share a set of POS with multiple vendors to interact with customers.

- **Customer**
  - Party who subscribes to offers through a SAP ODC server (not mandatory).
  - Party who uses Prepaid Accounts for payment or loyalty management.
  - Party who consumes/uses services provided by the vendors.
  - A customer can be an operator's client or a vendor's client or both.

Architecture



SAP ODC is defined through the following functional architecture. 3 main modules compose this architecture:

- SAP-ODC Server: the main access point for the offer cataloguing and the provisioning processes. This module is based on SAP CC data models that have been enhanced for supporting the embedded aspects.

- MER (Mobile Embedded Rater): this functional module is embedded on the device. It's a container which hosts the offers and Prepaid Accounts which have been subscribed by a given customer. At this time, this module is implemented into a Smart card or a Secured Element which

can support Javacard technologies. However some ODC implementations can also provide non-secure MER able to run on Android.

- POS (Point Of Service / Sale): this is mediation module in charge of sending the Chargeable Items (CDRs/EDRs) and/or debit/credit the Prepaid Account according to the service consumption. The POS can be implemented by a vendor or an operator. This module sends the CDR/EDR to the MER to be rated.

- The external system module represents different functional services which are out of scope from SAP SAP-ODC but can be interfaced with. By example: online payment system, billing/invoicing system, POS management module, CRM, vendor portal, app store, service platform …

The connections between modules can be implemented on different protocols such as OTA, NFC, ISO, ADSL, etc. (for OTA connections SAP ODC should be interfaced with a TSM platform for being able to communicate with the secure elements located on mobile except if operator is the card issuer AND if the DIRECT deployment mode is used).

Main processes

The main workflows can be summarized through the 2 main following use-cases:

Charging (credit/debit) an account

The following processes have to be performed with this use case:

Vendor offline account provisioning and refilling

"A referenced (by the operator) vendor provisions a prepaid account for a customer"

- The customer accesses to the vendor portal or the operator portal aggregating several services through different media (web, from its mobile …).
- The vendor proposes an offer which is just made of a private payment mean (prepaid card, gift card or postpaid account) or of a balance for storing loyalty points.
- The customer subscribes to the offer, a subscriber account (SUAC) is then created (if it doesn't still exist) on the ODC server between the vendor and the customer and the related account (prepaid or postpaid) is created into the SUAC.
- The SAP-ODC server uploads the new subscriber account (if necessary) and the account in the customer's MER.
- For prepaid account types the user is then invited to refill the balance by using an external payment mean.
- Then, the ODC server (if the payment is validated), synchronizes the refill with the offline customer's account (this sub-process can be performed in a separated / standalone process).

(*) Please note that several accounts can be created in given SUAC that defines the business relationship between a vendor and a customer.

Credit or debit the customer's offline account

"An existing offline account is used as a payment instrument or as a loyalty point counter".

- The customer accesses to a service he wants to consume
- Before delivering the service/good the vendor wants to perform the payment by debiting the customer's prepaid account or the vendor wants to add loyalty points on the customer balance
- A communication link is created between the POS and the MER
- The POS calls the debit or credit API on the MER by specifying the reference of the account that must be debited / credited.
- Then the amount is subtracted or added to the account balance amount after having checked the currency compliance.
- According to the account response (OK or insufficient balance if the empty limit has been reached) the service is delivered or not.

Working with on-device offers able to charge offline accounts

The following processes have to be performed with this use case:

Vendor offer building process

"A referenced (by the operator) vendor wants to add a new offer into its catalogue about a new or an existing service"

- The vendor uses the ODC core tool to define its offer ("embeddable and compatible with the prepaid service).
- This offer will be defined according to the Chargeable Items description which will be sent by each POS representing the service.
- When the offer is ready, the vendor registers the offer onto the SAP ODC server.
- He can publish the service onto its own (Web) portal or on a (Web) portal provided by the operator aggregating other services (provided by other vendors).
- Then the offer is ready for the provisioning process.

A customer wants to subscribe to the "embed-able" vendor offer (*)

"The customer chooses the service and the embed-able offer and subscribes to it"

- The customer accesses to the vendor portal or the operator portal aggregating several services through different media (web, from its mobile …).
- The vendor's offers are then listed and explained
- The customer chooses the appropriate one and personalizes it with his parameters (if required).
- He's invited to pay it through an external payment system (optional)
- Then, the portal creates on SAP-ODC server the subscription with customer's parameters and the associated account if the customer is new.
- SAP-ODC server generates (through the compiler) the embedded version of this subscription.
- If the client is a new one (for this vendor), the SAP-ODC Office creates a new "Subscriber Account" (with the required balances) into the customer's MER.
- The SAP-ODC server uploads the new subscription into the customer's MER.
- SAP-ODC Office activates this new subscription into the MER.

(*) the above sequence is an interactive scenario but the offer subscription can be totally handled, automatically, by the vendor through some provisioning integration code.

A customer consumes a service:

- The customer accesses to a service he wants to consume
- A communication link is created between the POS and the MER
- The POS sends a CDR/EDR corresponding to the service with additional information such as the Vendor ID, consumption date, to the customer's MER.
- The MER retrieves from the Vendor ID the right Subscriber Account (the account corresponding to this vendor into the MER).
- If no subscriber account is found the chargeable event is rejected and the service is not delivered.
- When the Subscriber Account is found, the CDR/EDR's name is used to identify the subscription which is involved by this service consumption.
- Then, the subscription's rating process is triggered (a price and/or counter impacts are calculated) and this process returns an authorization (or a rejection) to the POS after having charged the price on prepaid balances (if required by the charging plan).
- According to the rating process response, the service is delivered or not.

Technologies

ODC runs on Javacard / Global Platform technologies, mainly for security reasons, to guarantee the integrity of the code.

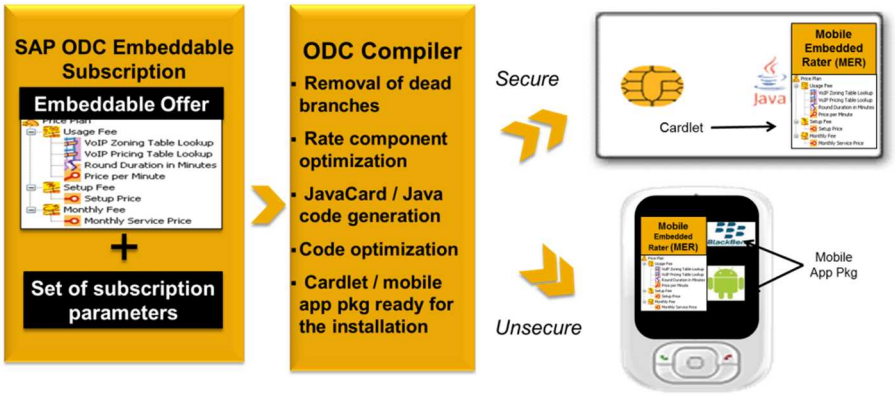
Javacard and GP technologies are mainly implemented by inexpensive components called Secure Element. Several form factors exist for SE: SIM card, micro SD card, embedded SE, dongle, key fob, … .

By combining those Secure Element with the NFC (Near Field Communication) technology, the connection with the Point of Sale/Service can now be done in a secure way with a contactless/proximity interaction and a very efficient pairing time (less than 0.2 second).

The counterpart is that smart card/secure element resources are not sufficient for implementing and running a generic rating engine on it.

To solve this issue, a compiler has been developed to produce some specific rating engines for each customer's subscription instead of developing a generic rating engine as it is implemented into the centralized SAP CC version.

On ODC server, each subscription to an offer will be transformed by the compiler into a dedicated and optimized applet which will be installed on the end user's Secure Element.

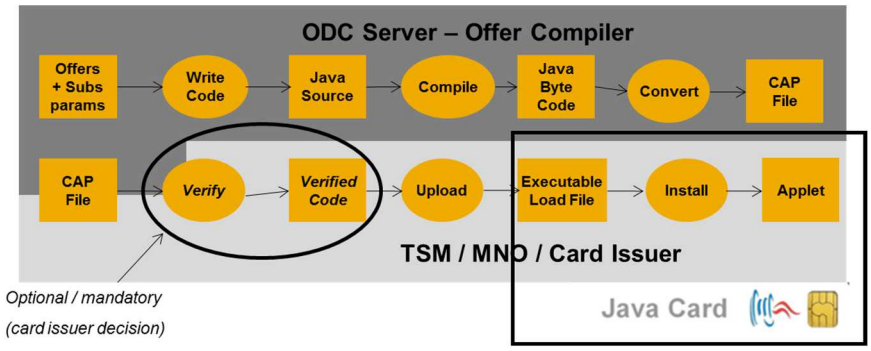


The typical steps when creating a Java Card application are:

1. Write the Java source.
2. Compile your source.
3. Convert the class files into a Converted Applet (CAP) file.
4. Verify that the CAP is valid; this step is optional.
5. Install the CAP file.

The 3 first steps are currently performed by the ODC compiler.

The step 5 can be performed by ODC or delegated to a third party (TSM) depending of the business case and the card issuer.

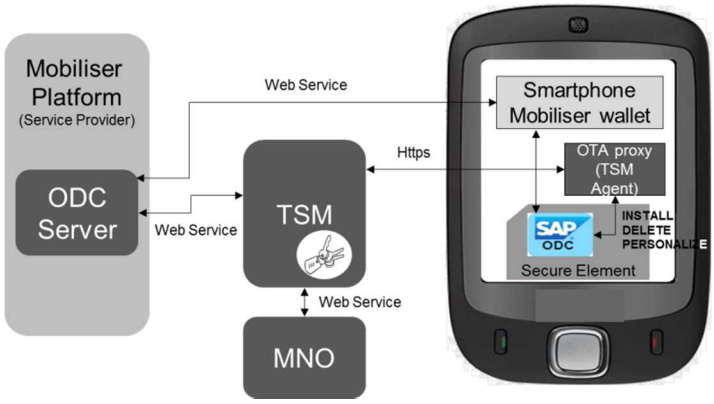


Remarks regarding the ODC integration with Mobiliser

Regarding the installation step (step 5 above), ODC with Mobiliser supports 2 modes to deploy/install the Cardlet (including the MER) on the Secure Element.

- DIRECT mode: the operator of the Mobiliser platform is the SE issuer and knows the key set for managing the SE. With this mode, if the SE is compliant, it can be directly managed by the ODC server in Mobiliser Core platform.

- DELEGATED mode: the operator of the Mobiliser platform is not the SE issuer and requires the services of a TSM (Trusted Service Management) to manage the ODC applet on the SE. With this mode, integration with the TSM Web Services that will be selected by the Mobiliser Operator is required.



ODC MER Architecture Overview

The Mobile Embedded Rater is a card applet installed into a reserved Security Domain implemented into a Javacard (see Global Platform specifications). This applet plays the role of a container that manages some subscriber accounts and the accesses to the business logics (Subscription applets) that can be installed dynamically into the same security domain.

Data Model

A Subscriber Account represents the relationship between a Merchant and a Customer. A Subscriber Account contains one or several accounts that can be prepaid (with a balance) or external (just a reference to an external account). Due to limitations imposed by the SE, the number of Subscriber Accounts which can be installed in a MER is limited and will depend on the SE profile (memory size). It's difficult to give some figures. By example on a referenced SE, the CredenSE from Device Fidelity with 70 kBytes can host about 10 Subscriber Accounts with a single Prepaid Account per Subscriber Account.

A Prepaid Account represents a monetary balance that can be credited, debited (from the POS) or refilled (from the ODC server). An empty limit can be defined (default is 0) for each Prepaid Account. Several Prepaid Accounts can be defined in a Subscriber Account and at least one Prepaid Account must be defined as default in a given Subscriber Account. This default Prepaid Account will be automatically selected if no Charging Plan is defined or if a Prepaid Account referenced by a charging plan cannot be found. Moreover, a non-monetary currency (the POINT code PNT) is added for managing loyalty policy.
In addition, considering that a Prepaid Account is embedded in the SE (that is a constrained architecture) some limitations are applied on balance values that must be comprised between -9999.9999 and

9999.9999. At last, the number of Prepaid Account per Subscriber Account is limited to 3 items by default (this limitation can be changed in SE profiles).

A Subscription represents an offer that has been subscribed by a user. The Subscription mainly contains the business logic, the offer parameters and the counters. A Subscription installed with the MER is always a copy of a Subscription that has been created into the ODC server except for the counter values that are updated / modified on the device itself.
At last, considering that a subscription is installed on the SE, some limitations exist on the following elements. Those limitations are controlled by the ODC Tool and by the server before recording an offer and a subscription into the ODC DB.
Limitations on offer:

- All strings are limited to 24 characters (extended ASCII)
- Number range is [-9999.9999 , 9999.9999]
- Max row count for a Translation Table : 100 (max column count is 15 – 5 inputs and 10 outputs-)
- Date range is [2000/1/1; 2099/12/31]
- In a given Price Plan the number of counters and the number of parameters is limited to 5 (each).
- The number of transient counters used in a price plan is limited to 3.
- Sub-offers are not supported.

Limitations on subscription (controlled by the ODC server at creation time):

- A Subscription cannot contain more than 10 Charge Activation.
- Translation Table row count is limited to 100 for all translation tables used by a subscription.
- The total number of counters implemented by a subscription is limited to 20.
- The total count of parameters used by a subscription is limited to 20.
- The total count of transient counters used by a subscription is limited to 5.

At last, the number of Subscriptions that can be deployed on a SE will depend of the size of each Subscription and the available remaining memory in the SE. For information the smallest Subscription size is around 4Kbytes and the largest can take 60Kbytes.

Access and Subscription are very close. The Access identifies the subscription that must be used/triggered when a Chargeable Item is received by the MER. For performing this guiding process, the MER will use the serviceID and the merchantID.

Chargeable Item describes the usage of a service or a purchase. This data is send to the MER by the POS (Point Of Sale / Service) to be charged by the suitable subscription. This Chargeable Item contains some default properties and some additional user properties that can be defined by the pricing designer. Due to limitations imposed by the SE, only 7 user properties (date, number or string) can be added. This limitation is controlled by the ODC Tool when the Chargeable Items are defined (see *Chargeable Item Class* definition).

EToken is the result of a transaction which has been charged on a Prepaid Account. It provides a signature that guarantees the integrity of the transaction. EToken can be stored into the SE or/and returned to the POS. The number of eTokens that can be stored into the SE, depends of the remaining SE memory and the cryptographic method that has been selected. The size of a eToken is 113 bytes (with 3-DES) or 228 bytes (with RSA).

Remarks regarding the ODC integration with Mobiliser

In Mobiliser the notions of Subscriber Account and Prepaid Account are grouped under the notion of Stored Value Account (SVA). A SVA is issued by a merchant and owned by a customer. When a new SVA is created on a customer's device by a merchant, Mobiliser will check if a Subscriber Account already exists between the merchant and the customer and will create it if not. Then, this SVA will be used for purchasing services provided (only) by the referenced merchant. At last, a special SVA (called SHARED SVA), issued by the operator of the Mobiliser platform can be used for purchasing services/good proposed by all the merchants that are registered in the Mobiliser platform.

MER APIs & rating / charging processes overview

The MER provides 4 sets of APIs:

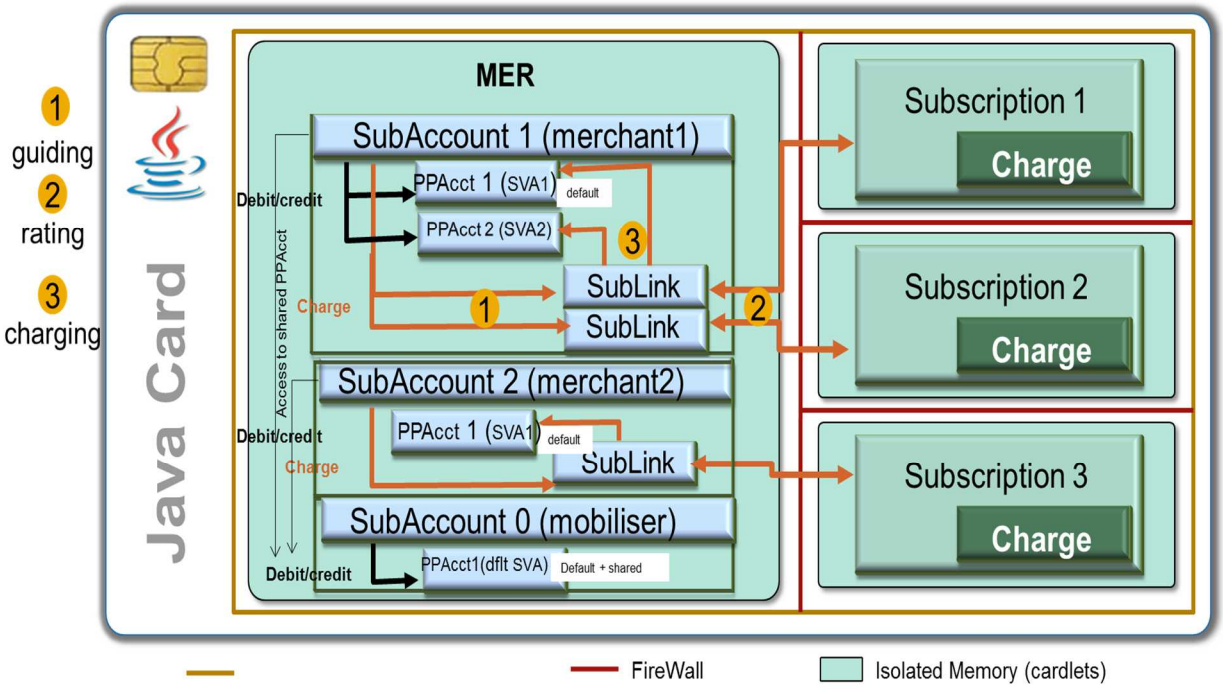
- POS API that allows a POS (Point of Service – NFC or not) to credit/debit a Prepaid Account or to charge consumption (by using NFC protocol by example) on dedicated business logic (if it exists).
- ADMIN API that allows the ODC Server (and only the ODC Server) to read/write/delete the data

into the MER (incl. the MER itself and the Subscription cardlets).

- VIEWER API that allows mobile APK (such as Smartphone Mobiliser) to read the data that is stored into the MER Applet.
- READ API that allows a merchant/vendor application deployed on a POS to read the merchant's data that is stored into the MER Applet.

The following schema shows how data is organized inside the Security Domain used by the MER and Subscription. At last the reader can see how the POS APIs directly impact the SVA (debit/charge) and trigger the global charging process as following:

1. The merchant POS builds and ciphers (thanks to a key previously generated by the ODC server) the Chargeable Item related to the usage.
2. The POS sends to the MER a (ciphered) Chargeable Item with the consumption date, the merchant ID and the service ID.
3. Thanks to the merchant ID the MER will check that a subscriber account with this merchant already exists.
4. The MER deciphers the Chargeable Item thanks to the (symmetric) merchant key that is recalculated inside the MER.
5. If the Chargeable Item cannot be deciphered, it is rejected by the MER.
6. Thanks to the service ID and the access the Chargeable Item is guided towards the associated subscription for the rating process.
7. During this rating process, the subscription will execute the business logic, impact the counters and generate one or several transactions.
8. After having defined the ID of the accounts (prepaid or external) on which each transaction will be charged, the set of transactions is returned to the MER by the subscription.
9. Each transaction is then charged on the related account (debited from the Prepaid Account or marked with the external account ID).
10. If the charging process fails a rejection result is the returned to the POS.
11. If required, those transactions are signed by the MER for building one or several some eTokens.
12. At last, a (positive/negative) response is returned to the POS that contains or not (based on charge() call parameters) the eTokens.



At last, when the debit/credit process is directly triggered by the merchant POS, the code of the targeted SVA must be passed as parameter by the merchant in addition to the amount to credit/debit.

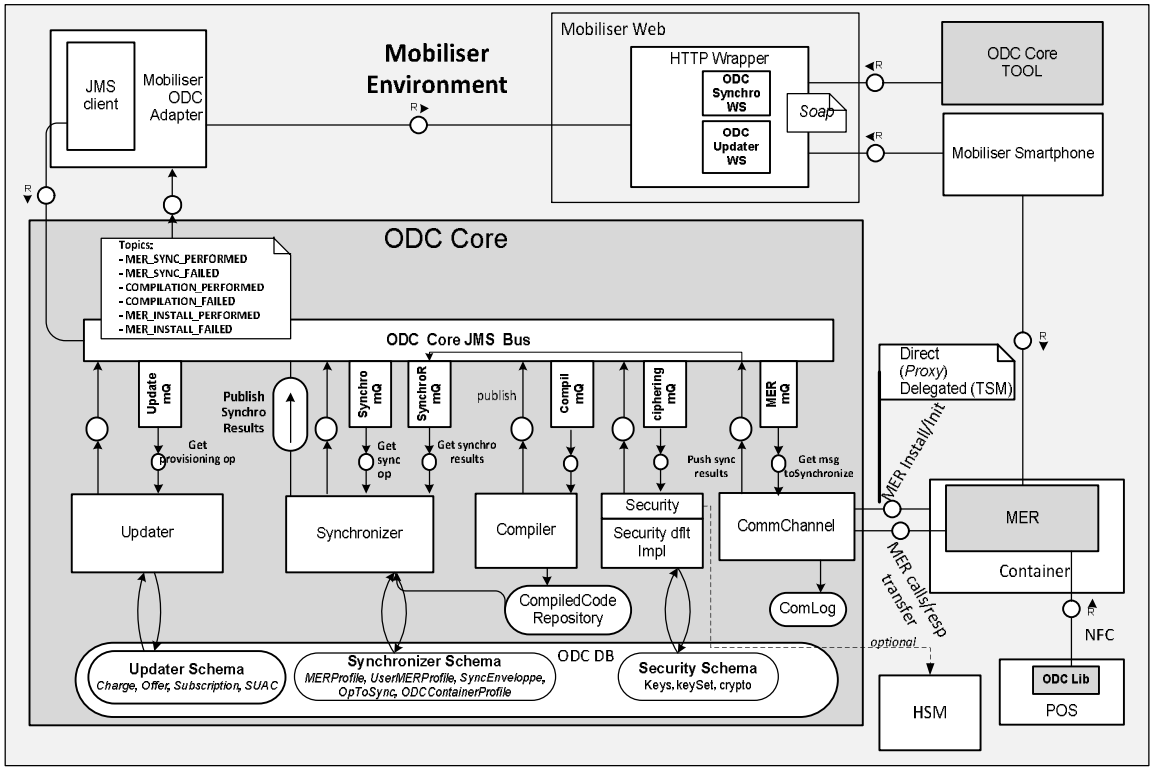
Then if a Subscriber Account involving the merchant is found, the MER will attempt to debit the specified Prepaid Account if it exists. Otherwise, the MER will try to debit the default account. If the amount cannot be (totally) charged on the account (default or not) or if an amount is out of range or if the no Subscriber Account has been found, the transaction is discarded and the process returns a rejection code to the POS.

Remarks regarding the ODC integration with Mobiliser

As explained above, the integration has introduced the notion of SHARED SVA (unique per MER). With the Mobiliser integration, if the debit/credit process cannot find a Subscriber Account for the specified merchant, the process will try to debit/credit the SHARED SVA that is implemented into a different Subscriber Account owned, in that case, by the Mobiliser platform representative/operator (that will play the merchant role).

Each time the SHARED SVA will be charged, an eToken identifying the transaction date, the customer, the merchant, the SVA code and the amount, will be automatically generated for feeding the clearing process on the Mobiliser platform.

ODC Server architecture overview



The ODC server is built around a Java Message Service that is used by 5 functional core modules:

- Updater: this module is used to provision all the business data required by ODC.
- Synchronizer: the Synchronizer is the orchestration module that manages the (asynchronous) interactions with the Mobile Embedded Rater
- Compiler: This module is able to transform the subscriptions provisioned by the Updater into Javacard code that can be installed with the MER into the SE Container (a Security Domain as defined by GP).
- Security: as ODC server is able to store and use the Secure Element keys (DIRECT mode), this module manages the reliable storage of all ODC keys and the execution of the cryptographic algorithms that are required by other modules. This module which defines a functional interface can be connected to an external HSM solution (optional) to increase the security.
- CommChannel: this module is the transportation layer that organizes and sends all the data (APDU) to the MER and SE container. This module don't access directly to the SE but uses some

WS from TSM (DELEGATED mode) or uses the connections Mobiliser already have with the Smartphone Mobiliser (by using a push OTA notification, SMS, GCM, …). With this last case, Smartphone Mobiliser is used as a proxy to transmit ciphered data to the MER/SE Container.

To access to ODC core, an ODC Adapter which implements a JMS client, provides to Mobiliser a simplified API and builds the notion of Mobiliser SVA on concept implemented by ODC.

At last, the POS (Point of Service / Point of Sale) interacts with the MER by using NFC and ODC libs.

Security architecture

Based on Javacard / Global platform specifications, the access to the SE's administration is only authorized to the SE issuer (or trusted third party - TSM). The SE's administration requires a dedicated key set for creating, by example, the security domain (SD) that will be used by ODC MER. This feature is out of scope of ODC. To deploy the MER on the SE via the DIRECT mode, a security domain must have been previously created and the key set for acceding to this SD (see ODC_S-ENC, ODC_S-MAC, and ODC_DEK below) must be recorded into the ODC server.

All the communications with the MER must be ciphered for getting a response from it.

We can list 3 types of interactions with the MER:

- ODC Server / MER interactions: security is based on a standard Global Platform secure channel (based on SCP02) that will cipher all the exchanges between the ODC server and the MER (symmetric encryption). To implement this secure channel, the ODC server will use the following GP key set: ODC_S-ENC, ODC_S-MAC, and ODC_DEK preinstalled into the SE. Depending on the pre-personalization required by the SE issuer to the SE manufacturer, the keyset can be unique for each Secure Element (SE).

  When, the DIRECT mode is used (see above), those key sets are ciphered by a RSA public key and stored into the ODC database. They will be deciphered (thanks to a private key) and used by the ODC to implement the secure channel. When the DELEGATED mode is used, those keys and the secure channel are managed by the TSM and ODC will ask the TSM to establish the secure channel.

- POS / MER interactions: security is based on a single ODC private key (called MPcK – MER private charging key -) and a vendor alea (vALEA) generated by ODC for each new merchant (identified by a hash code vID). As an isolation mechanism is mandatory between the different subscriber accounts (to avoid a merchant A charging a SVA issued by a merchant B), for each new merchant, the ODC server will generate a vendor charging key (*VcK = DESede[MPCK, vALEA + vID]*). VcK is used by the merchant's POS to charge/debit/credit the customer's subscriptions/SVA. As the MPcK is installed into a MER during its personalization phase, when a subscriber account is created into the MER, the vendor vALEA is also pushed into the MER. Then, the MER will be able to communicate in a secure way with the related merchant's POS.

  Moreover, in a similar way, for allowing a merchant to read (see above READ API) its own data stored into a customer's MER, a vendor read key (VrK) is generated by ODC from a dedicated ODC private key (called MPrK – MER Private readKey) and the vendor alea. Like the VcK, the VrK is also delivered to the merchant.

- Mobile APP / MER interactions: to access to the READ API provided by the MER, the mobile APP will have to request a MER viewer Key (MvK) from the MER (to cypher the communication). This key is randomly generated (when requested) by the MER and then stored into the mobile APP and the MER. By example, this key is requested by Smartphone Mobiliser when a new MER is deployed or when the application tries to access to the MER for the first time. Each time a viewer key is requested, the MER will invite the user to enter its MER PIN code before generating the MvK (by using DESede).

  This viewer key prevent non authorized applications to access to the customer's data and prevent authorized applications to access to the MER data from other customers when the SE is moved from a mobile to another one.

In addition to the above secure mechanisms used for communicating with the MER, the MER uses a private key (MPsK) for signing the eTokens.

At last, the MER provides a wallet PIN mechanism for asking the customer to enter its PIN on the

following configurable (by the customer or ODC server) actions/criteria:

- viewer key generation
- transaction amount threshold
- expiration of inactive period
- max count of authorized transaction

To conclude with the security architecture, ODC server implements a separated security module that stores the different key set required for managing the SEs (DIRECT mode). This module is also used for generating the vendor keys and running the cryptographic algorithms. Optionally, this module can be replaced by a HSM solution to strengthen the security level.

However, without a HSM, all the keys that are stored into the ODC database must be ciphered to avoid being read directly. By example, that's what is has done with the Mobiliser Integration: all ODC keys have been ciphered thanks to a RSA public key and the related private key to decipher has been stored into the Mobiliser 's key store.

Point of Sale / Service API description

Tables may include a "MOC" column meaning "Mandatory/Optional/Conditional". This column specifies the obligation of presence of the data in the function or in the message. The following definitions apply to these terms:
• Mandatory (M): Means that an entry must be supplied
• Optional (O): Means that an entry can be supplied, but is not required to be supplied.
• Conditional (C): Means that the usage of an entry is dependent upon a particular condition.

Basic Types definition

String24	String with a maximum length set to 24 ASCII_CHAR.
Date5	5 bytes encoding date:
Decimal5	A 5 bytes decimal with up to 4 decimal digits strictly greater than -10,000 and strictly lower than 10,000.
HexByte2	A byte in hexadecimal (2 chars): example “2A”
Short2	2 bytes short number

Charging APIs

DEBIT_ACCOUNT_MER_COMMAND: DebitAccount()

This API allows a POS (owned by a service provider) to debit a Prepaid Account and assign the debited amount to a payee ID (previously provisioned in an external system). Most of time the Service Provider and the payee ID are the same party, but sometimes a service provider can collect money on behalf of a third party.

Input data name	Description	Type	MOC
SERVICE_ PROVIDER_CODE	The code that identifies the service provider and consequently the Subscriber Account into the MER (only 1 or zero Sub. Acct. per merchant).	**String24**	M
ACCOUNT_CODE	The code of the (prepaid) account that must be debited. For a given service provider, this code is unique and references an account provisioned on the end-user’s device. If not set, the default account is debited.	**String24**	O
AMOUNT	The amount that must be debited from the Prepaid Account	**Positive Decimal5**	M
CURRENCY	The ISO 4217 currency 3 letters code. Please note that 2 additional non- monetary codes are available: XXX for no currency and PNT used for POINTS.	**ASCII_CHAR[3]**	M
DATE	The debit date. Most of time the current date of the POS that performs the debit.	**Date5**	
PAYEE_ID	The ID of the party who receives the amount debited from the customer’s Prepaid Account.	**String24**	M
PROOF_POLICY	Indicates the policy for generating the tokens	**Byte with one of the following values: PROOF_NOT_REQUIRED = 0; PROOF_RETURNED = 1; PROOF_STORED = 2; PROOF_STORED_AND_RETURNED = 3;**	M

Remarks regarding the ODC integration with Mobiliser

The SERVICE PROVIDER CODE is mapped on the merchant name and the PAYEE ID input data is mapped on the merchant ID as defined by Mobiliser.

The PROOF_POLICY is always set to STORED_AND_RETURNED: the tokens are always generated, returned to the caller and kept into the MER.

CREDIT_ACCOUNT_MER_COMMAND: CreditAccount()

This API allows a POS (owned by a service provider) to refund/credit a Prepaid Account and assign the credited amount to a payer ID (previously provisioned in an external system). Most of time the Service Provider and the payer ID are the same party, but sometimes a service provider can refund money on behalf of a third party.

Input data name	Description	Type	MOC
SERVICE_ PROVIDER_CODE	The code that identifies the service provider and consequently the Subscriber Account into the MER (only 1 or zero Sub. Acct. per merchant).	**String24**	M
ACCOUNT_CODE	The code of the (prepaid) account that must be credited. For a given service provider, this code is unique and references an account provisioned on the end-user's device. If not set, the default account is credited.	**String24**	O
AMOUNT	The amount that must be credited on the Prepaid Account	**Positive Decimal5**	M
CURRENCY	The ISO 4217 currency 3 letters code. Please note that 2 additional non-monetary codes are available: XXX for no currency and PNT used for POINTS.	**ASCII_CHAR[3]**	M
DATE	The credit date. Most of time the current date of the POS that performs the credit.	**Date5**	
PAYER_ID	The ID of the party who credits the amount on the customer's Prepaid Account.	**String24**	M
PROOF_POLICY	Indicates the policy for generating the tokens	**Byte with one of the following values:** **PROOF_NOT_REQUIRED = 0;** **PROOF_RETURNED = 1;** **PROOF_STORED = 2;** **PROOF_STORED_AND_RETURNED = 3;**	M

Remarks regarding the ODC integration with Mobiliser

The SERVICE PROVIDER CODE is mapped on the merchant name and the PAYER ID input data is mapped on the merchant ID as defined by Mobiliser.

The PROOF_POLICY is always set to STORED_AND_RETURNED: the tokens are always generated, returned to the caller and kept into the MER.

DEBIT_ACCOUNT_MER_RESPONSE & CREDIT_ACCOUNT_MER_RESPONSE: DebitAccount() / CreditAccount() Output Result

Output data name	Description	Type	MOC
STATUS	The status of the debit operation	**Byte with one of the following values:**	M

		STATUS_DEBIT_OK = 0; **STATUS_DEBIT_REJECTED = 1;** **STATUS_CREDIT_OK = 2;** **STATUS_CREDIT_REJECTED = 3;**	
REJECTION_CODE	If the status is rejected, it gives the reason of the rejection	**Byte with one of the following values :** **REJECTION_NONE = 0;** **REJECTION_ACCOUNT_INSUFFICIENT_BALANCE = 1;**	C
TOKEN	The signed transaction if the status is OK	**Set if the PROOF_POLICY differs from PROOF_NOT_REQUIRED** *(See Token Type description below)*	C

CHARGE_MER_COMMAND: Charge ()

This service allows the user to perform a business

Input data name	Description	Type	MOC
SERVICE_ PROVIDER_CODE	The code that identifies the service provider and consequently the Subscriber Account into the MER (only 1 or zero Sub. Acct. per merchant).	**String24**	M
SERVICE_CODE	The code that identifies the service that will be used by the MER for retrieving the access (that points on the subscription)	**String24**	M
CURRENT_DATE	The current DATE as returned by the POS.	**Date5**	M
CONSUMPTION_DATE	The date of the consumption (often the same than CURRENT_DATE for real time services.	**Date5**	M
PROOF_POLICY	Indicates the policy for generating the tokens	**Byte with:** **PROOF_NOT_REQUIRED = 0;** **PROOF_RETURNED = 1;** **PROOF_STORED = 2;** **PROOF_STORED_AND_RETURNED = 3;**	M
BLANK_CHARGE_FLAG	Flag for blank charge Blank charge doesn't impact counters and Prepaid Accounts.	**Byte with one of the following values:** **0: CHARGE (DFLT); 1: BLANK CHARGE**	M
CHARGEABLE_ITEM	The consumption data record that will be used as an input parameter for executing the related business logic.	**Chargeable Item Type**	M

Remarks regarding the ODC integration with Mobiliser

The SERVICE PROVIDER CODE is mapped on the merchant name as defined in Mobiliser.

The PROOF_POLICY is always (internally) set to STORED_AND_RETURNED: the tokens are always generated, returned to the caller and kept into the MER.

CHARGE_MER_RESPONSE: The Charge () Output Result

Output data name	Description	Type	MOC
PURCHASE_ORDER	Contains the result of the charging process.	**Purchase Order Type** *(See Purchase Order description below)*	M

TLV_TOKEN: Token Type description

Token field name	Description	Type	MOC
TOKEN_TYPE	3 types of token resulting of following API calls: - Charge() - Debit() / Credit() - P2P() (not yet used)	**Byte with one of the following values:** **TOKEN_TYPE_CHARGE = 0;** **TOKEN_TYPE_DEBIT_CREDIT = 1;** **TOKEN_TYPE_P2P = 2;**	M
CHARGE_TYPE	The token represents a customer's balance's debit or a balance's credit.	**CHARGE_TYPE_DEBIT = 0;** **CHARGE_TYPE_CREDIT = 1;**	M
DATE	The transaction date	**Date**	M
AMOUNT	The amount that must be credited on the Prepaid Account	**Positive Decimal5**	M
CURRENCY	The ISO 4217 currency 3 letters code. Please note that 2 additional non-monetary codes are available: XXX for no currency and PNT used for POINTS.	**ASCII_CHAR[3]**	M
ACCOUNT_CODE	The code of the account that that has been charged.	**String24**	M
PAYEE_ID	The ID of the party who is the "natural" amount receiver (even if the merchant refunds the client – credit charge -, the payee ID will stay the merchant ID)	**String24**	M
PAYER_ID	The ID of the party who is the "natural" amount emitter (even if the customer refunds the client – credit charge -, the payer ID will stay the customer ID)	**String24**	M
SIGNATURE	The signature of the transaction validating the above data.	**HexByte2[128]** with the default signature method (SHA1withRSA) **HexByte2[8]** if the DESedeMac64withISO7816-4Padding is used.	M
TOKEN_ID	Internal token identifier	**Short2**	M

Remarks regarding the ODC integration with Mobiliser

The PAYEE_ID field is always mapped on the merchant ID and the PAYER_ID is always mapped on the customer ID.

TLV_CHARGEABLE_ITEM: Chargeable Item Type

Field name	Description	Type	MOC
CHARGEABLE_ITEM_ID	The hash value of the Chargeable Item Name (as defined by the user in the ChargeableItemClass). The hash() function to use is described below.	**Short2**	M
PROPERTY_LIST	The list of the properties attached to the MER	*(See Property Type below)*	M

TLV_PROPERTY: Property Type

Field name	Description	Type	MOC
PROPERTY_ID	The hash value of the Property Name (as defined by the user in the ChargeableItemClass). The hash() function to use is described above.	**Short2**	M
PROPERTY_TYPE	The type of the property	*One of the following values:* **TYPE_NUMBER : 0** **TYPE_STRING : 1** **TYPE_DATE : 2**	M
PROPERTY_VALUE	The value of the property.	**Decimal5 or String24 or Date5**	M

TLV_PURCHASE_ORDER: Purchase Order Type

Field name	Description	Type	MOC
OFFER_CODE	The code of the offer that has generated the purchase order.	**String24**	M
SERVICE_CODE	The code of the service that has called the charge() api.	**String24**	M
CHARGE_DATE	Charging date	**Date5**	M
ORDER_STATUS	The result status	**Byte with one of the following values:** **ORDER_STATUS_PAID = 1;** **ORDER_STATUS_FREE = 2;** **ORDER_STATUS_REJECTED = 3;**	M
REJECTION_CODE	The reasons in case of fail.	**OK = 0x0000;** **PREPAID_INSUFFICIENT_BALANCE=0x0001;** **PREPAID_BALANCE_CLOSED = 0x0002;** **PREPAID_BALANCE_LOCKED = 0x0003;** **FORBIDDEN_CHARGE = 0x0010;** **NO_ACCESS = 0x0011;** **CHARGEABLE_ITEM_ID_INVALID = 0x0012;** **SUBSCRIPTION_OR_CHARGE_EXPIRED = 0x0020;** **SUBSCRIPTION_OR_CHARGE_NOT_YET_EFFECTIVE = 0x0021;** **DEFAULT_ERROR = 0xFFFF;**	M
REJECTION_REASON	The (business) reason when the NO_ACCESS is returned as REJECTION_CODE	**String24**	C
TOKEN_LIST	List of TOKEN returned by the business logic (max is 10 tokens)	**Set if the PROOF_POLICY in charge() differs from PROOF_NOT_REQUIRED** *(See Token Type description above)*	C

ODC hash() function

This function is used for hashing readable code into short because due to secure element memory limitation, all the object codes are transformed into Id (short) thanks to the following function.

All reference to hash() function in this document is based on the following algorithm.

```
public static short hash(String key) {
  int hash = 0;
  if ((key != null) && (key.length() > 0)) {
     char b[] = key.toCharArray(); // unicode based
     int i = 0;
     int length = b.length;
     while (i < length) {
        hash = (hash * 131) + b[i++];
     }
  }
 // Make it a positive int and return a short
 hash = hash & 0x0000FFFF;
 return (short) hash;
}
```

ODC MER PIN Checking

ODC provides an API (CheckMERPIN()) which authorizes the POS to ask the MER PIN number to the end user. See the TLV APDU description in the next sections.

Card ISO7816 Overview

Introduction

ISO7816 defines 4 parts:
- Part1: Physical Characteristics and Integrated Circuit Cards
- Part2: Dimensions and Location of the Contacts
- Part3: Electronic Signals and Transmission Protocols
- Part4: Inter-industry Commands for Interchange

We won't describe the 3 first parts that defines H/W standards.
We will just present an overview of the last part that will help us to understand how the C-APDU (Command Application Protocol Data Unit) and the R-APDU (Response) defined by the MER Charging and Read APIs must be used.

This section is based on the ISO standard and contains some extracts of the standard document (see www.cradwerk.com).

The part 4 of ISO/IEC 7816 smart card standard specifies:

- the contents of messages, commands, and responses transmitted by the interface device to the card and conversely,
- the structure and content of the historical bytes sent by the card during the answer to reset,
- the structure of files and data, as seen at the interface when processing inter-industry commands for interchange,
- access methods to files and data in the card,
- methods for secure messaging,
- Access methods to the algorithms processed by the card. It does not describe these algorithms.

It does not cover the internal implementation within the card and/or the outside world.

APDU Message Structure

Definitions

For the purposes of this part of the ISO/IEC 7816, the following abbreviations apply:

APDU	Application protocol data unit
ATR	Answer to reset
CLA	Class byte
INS	Instruction byte
P1-P2	Parameter bytes
Lc	Command length
Le	Length expected
SW1-SW2	Status bytes
TLV	Tag length value
'0'-'9' and 'A'-'F'	The sixteen hexadecimal digits
(B1)	Value of byte B1
B1||B2	Concatenation of bytes B1 (the most significant byte) and B2 (the least significant byte)
#	Number

Command APDU definitions

Illustrated by figure below, the command APDU defined in this part of ISO/IEC 7816 consists of

- a mandatory header of 4 bytes (CLA INS P1 P2),
- a conditional body of variable length

	Header (req)				Body (opt)		
Case 1: No cmd data & No resp required	CLA	INS	P1	P2			
Case 2: No cmd data & Yes resp required	CLA	INS	P1	P2	Le		
Case 3: Yes cmd data & No resp required	CLA	INS	P1	P2	LC	Data field	
Case 4: Yes cmd data & Yes resp required	CLA	INS	P1	P2	LC	Data field	Le

Table 1: The 4 structures of APDU

The number of bytes present in the data field of the command APDU is denoted by Lc.

The maximum number of bytes expected in the data field of the response APDU is denoted by Le (length of expected data). When the Le field contains only zeros, the maximum number of available data bytes is requested.

Decoding conventions for command bodies

In case 1, the body of the command APDU is empty. Such a command APDU carries no length field.

In cases 2, 3 and 4 the body of the command APDU consists of a string of L bytes denoted by B1 to BL as illustrated by table 2. Such a body carries 1 or 2 length fields; B1 is [part of] the first length field.

Command body
B1 B2 (L bytes)

Table 2: not empty body

In the card capabilities, the card states that, within the command APDU, the Lc field and Le field

- either shall be short (one byte, default value)
- or may be extended (explicit statement)

Consequently, the cases 2, 3 and 4 are either short (one byte for each length field) or extended (B1 is valued to '00' and the value of each length is coded on 2 other bytes).

Table 3 shows the decoding of the command APDUs according to the four cases defined in table 1 and table 2 and according to the possible extension of Lc and Le.

Conditions	Case
L=0	1

Decoding of the command APDUs

Decoding conventions for Le

If the value of Le is coded in 1 (or 2) byte(s) where the bits are not all null, then the value of Le is equal

to the value of the byte(s) which lies in the range from 1 to 255 (or 65535); the null value of all the bits means the maximum value of Le: 256 (or 65536).

The first 4 cases apply to all cards:

Case 1 - L=0 : the body is empty.

- No byte is used for Lc valued to 0
- No data byte is present.
- No byte is used for Le valued to 0.

Case 2 - L=1

- No byte is used for Lc valued to 0
- No data byte is present.
- B1 codes Le valued from 1 to 256

Case 3 - L=1 + (B1) and (B1) != 0

- B1 codes Lc (=0) valued from 1 to 255
- B2 to Bl are the Lc bytes of the data field
- No byte is used for Le valued to 0.

Case 4 - L=2 + (B1) and (B1) != 0

- B1 codes Lc (!=0) valued from 1 to 255
- B2 to Bl-1 are the Lc bytes of the data field
- Bl codes Le from 1 to 256
- shows the contents of the command APDU.

Response APDU definition

Illustrated by Table 4 (see also table 5), the response APDU defined in this part of ISO/IEC 7816 consists of

- a conditional body of variable length
- a mandatory trailer of 2 bytes (SW1 SW2)

Body	Trailer
[Data field]	SW1 SW2

The number of bytes present in the data field of the response APDU is denoted by Lr.

The trailer codes the status of the receiving entity after processing the command-response pair.

NOTE - If the command is aborted, then the response APDU is a trailer coding an error condition on 2 status bytes.

Coding conventions for command headers, data fields and response trailers

Code	Name	Length	Description
CLA	Class	1	Class of instruction
INS	Instruction	1	Instruction code
P1	Parameter 1	1	Instruction parameter 1
P2	Parameter 2	1	Instruction parameter 2
Lc field	Length	variable 1 or 3	Number of bytes present in the data field of the command
Data field	Data	variable=Lc	String of bytes sent in the data field of the command
Le field	Length	variable 1 or 3	Maximum number of bytes expected in the data field of the response to the command

Table 4 - command APDU contents

Code	Name	Length	Description
Data field	Data	variable=Lr	String of bytes received in the data field of the response
SW1	Status byte 1	1	Command processing status
SW2	Status byte 2	1	Command processing qualifier

Table 5 - response APDU contents

The subsequent clauses specify coding conventions for the class byte, the instruction byte, the parameter bytes, the data field bytes and the status byte. Unless otherwise specified, in those bytes, RFU bits are coded zero and RFU bytes are coded '00'.

Class byte

According to table 6 used in conjunctions with table 7, the class byte CLA of a command is used to indicate :

- to what extent the command and the response comply with this part of ISO/IEC 7816
- and when applicable (see table 7), the format of secure messaging and the logical channel number.

Value	Meaning
'0X'	Structure and coding of command and response according to this part of ISO/IEC 7816 (for coding of 'X' see table 7)
10 to 7F	RFU
8X, 9X	Structure of command and response according to this part of ISO/IEC 7816. Except for 'X' (for coding, see table 7), the coding and meaning of command and response are proprietary
AX	Unless otherwise specified by the application context, structure and coding of command and response according to this part of ISO/IEC 7816 (for coding of 'X', see table 7)
B0 to CF	Structure of command and response according to this part of ISO/IEC 7816
D0 to FE	Proprietary structure and coding of command and response
FF	Reserved for PTS

Table 6 - Coding and meaning of CLA

b4 b3 b2 b1	Meaning
x x -- --	Secure messaging (SM) format
0 x -- --	No SM or SM not according to 1.6
0 0 -- --	No SM or no SM indication
0 1 -- --	Proprietary SM format
1 x -- --	Secure messaging according to 1.6
1 0 -- --	Command header not authenticated
1 1 -- --	Command header authenticated (see 1.6.3.1 for command header usage)
-- -- x x	Logical channel number (according to 1.5) (b2 b1 = 00 when logical channels are not used or when logical channel #0 is selected

Table 7 - Coding and meaning of nibble 'X' when CLA='0X','8X','9X' or 'AX'

Instruction byte

The instruction byte INS of a command shall be coded to allow transmission with any of the protocols defined in part 3 of ISO/IEC 7816. Table 8 shows the INS codes that are consequently invalid.

b8 b7 b6 b5 b4 b3 b2 b1	Meaning
x x x x x x x 1	Odd values
0 1 1 0 x x x x	'6X'
1 0 0 1 x x x x	'9X'

Table 8 - Invalid INS codes

Table 9 shows the INS codes defined in this part of ISO/IEC 7816. When the value of CLA lies within the range from '00' to '7F', the other values of INS codes are to be assigned by ISO/IEC JTC 1 SC17.

Value	Command name
'0E'	ERASE BINARY
'20'	VERIFY
'70'	MANAGE CHANNEL
'82'	EXTERNAL AUTHENTICATE
'84'	GET CHALLENGE
'88'	INTERNAL AUTHENTICATE
'A4'	SELECT FILE
'B0'	READ BINARY
'B2'	READ RECORD(S)
'C0'	GET RESPONSE
'C2'	ENVELOPE
'CA'	GET DATA
'D0'	WRITE BINARY

'D2'	WRITE RECORD
'D6'	UPDATE BINARY
'DA'	PUT DATA
'DC'	UPDATE DATA
'E2'	APPEND RECORD

Table 9 - INS codes defined in this part of ISO/IEC 7816

Parameter bytes

The parameter bytes P1-P2 of a command may have any value. If a parameter byte provides no further qualification, then it shall be set to '00'.

Data field bytes

Each data field shall have one of the following three structures.

- Each TLV-coded data field shall consist of one or more TLV-coded data objects.
- Each non TLV-coded data field shall consist of one or more data elements, according to the specifications of the respective command.
- The structure of the proprietary-coded data fields is not specified in ISO/IEC 7816.

This part of ISO/IEC 7816 supports the following two types of TLV-coded data objects in the data fields :

- BER-TLV data objects
- SIMPLE-TLV data object

ISO/IEC 7816 uses neither '00' nor 'FF' as tag value.

Each BER-TLV data object shall consists of 2 or 3 consecutive fields (see ISO/IEC 8825 and annex D).

- The tag field T consists of one or more consecutive bytes. It encodes a class, a type and a number.
- The length field consists of one or more consecutive bytes. It encodes an integer L.
- If L is 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.

Each SIMPLE-TLV data object shall consist of 2 or 3 consecutive fields.

- The tag field T consists of a single byte encoding only a number from 1 to 254 (e.g. a record identifier). It codes no class and no construction-type.
- The length field consists of 1 or 3 consecutive bytes. If the leading byte of the length field is in the range from '00' to 'FE', then the length field consists of a single byte encoding an integer L valued from 0 to 254. If the leading byte is equal to 'FF', then the length field continues on the two subsequent bytes which encode an integer L with a value from 0 to 65535.
- If L in not null, then the value field V consists of consecutive bytes. If L is null, then the data object is empty: there is no value field.

The data fields of some commands (e.g. SELECT FILE), the value fields of the SIMPLE-TLV data object and the value field of the some primitive BER-TLV data objects are intended for encoding one or more data elements.

The data fields of some other commands (e.g. record-oriented commands) and the value fields of the other primitive BER-TLV data objects are intended for encoding one or more SIMPLE-TLV data objects.

The data fields of some other commands (e.g. object-oriented commands) and the value fields of the constructed BER-TLV data objects are intended for encoding one or more BER-TLV data objects.

NOTE - Before between or after TLV-coded data objects, '00' or 'FF' bytes without any meaning may occur (e.g. due to erase or modified TLV-coded data objects).

Important Note: ODC/MER doesn't use the BER-TLV encoding but only the SIMPLE-TLV.

Status bytes

The status bytes SW1-SW2 of a response denote the processing state in the card.

NOTE - When SW1='63' or '65', the state of the non-volatile memory is changed. When SW1='6X' except '63' and '65', the state of the non-volatile memory is unchanged.

Due to specifications in part 3 of ISO/IEC 7816, this part does not define the following values of SW1-SW2 :

- '60XX'
- '67XX', '6BXX', '6DXX', '6EXX', '6FXX'; in each case if 'XX'!='00'
- '9XXX', if 'XXX'!='000'

The following values of SW1-SW2 are defined whichever protocol is used (see examples in annex A).

- If a command is aborted with a response where SW1='6C', then SW2 indicates the value to be given to the short Le field (exact length of requested data) when re-issuing the same command before issuing any other command.
- If a command (which may be of case 2 or 4, see table 2) is processed with a response where SW1='61', then SW2 indicates the maximum value to be given to the short Le field (length of extra data still available) in a GET RESPONSE command issued before issuing any other command.

NOTE - A functionality similar to that offered by '61XX' may be offered at application level by '9FXX'. However, applications may use '9FXX' for other purposes.

Table 10 completed by tables 11 to 16 shows the general meanings of the values of SW1-SW2 defined in this part of ISO/IEC 7816. For each command, an appropriate clause provides more detailed meanings.

Tables 11 to 16 specify values of SW2 when SW1 is valued to '62', '63', '65', '68', '69' and '6A'. The values of SW2 not defined in tables 11 to 16 are RFU, except the values from 'F0' to 'FF' which are not defined in this part of ISO/IEC 7816.

SW1-SW2	Meaning
	Normal processing

'9000'	No further qualification
'61XX'	SW2 indicates the number of response bytes still available (see text below)
	Warning processings
'62XX'	State of non-volatile memory unchanged (further qualification in SW2, see table 11)
'63XX'	State of non-volatile memory changed (further qualification in SW2, see table 12)
	Execution errors
'64XX'	State of non-volatile memory unchanged (SW2='00', other values are RFU)
'65XX'	State of non-volatile memory changed (further qualification in SW2, see table 13)
'66XX'	Reserved for security-related issues (not defined in this part of ISO/IEC 7816)
	Checking errors
'6700'	Wrong length
'68XX'	Functions in CLA not supported (further qualification in SW2, see table 14)
'69XX'	Command not allowed (further qualification in SW2, see table 15)
'6AXX'	Wrong parameter(s) P1-P2 (further qualification in SW2, see table 16)
'6B00'	Wrong parameter(s) P1-P2
'6CXX'	Wrong length Le: SW2 indicates the exact length (see text below)
'6D00'	Instruction code not supported or invalid
'6E00'	Class not supported
'6F00'	No precise diagnosis

Table 10 - Coding of SW1-SW2

SW2	Meaning
'00'	No information given
'81'	Part of returned data may be corrupted
'82'	End of file/record reached before reading Le bytes
'83'	Selected file invalidated
'84'	FCI not formatted according to 1.1.5

Table 11 - Coding of SW2 when SW1='62'

SW2	Meaning
'00'	No information given
'81'	File filled up by the last write
'CX'	Counter provided by 'X' (valued from 0 to 15) (exact meaning depending on the command)

Table 12 - Coding of SW2 when SW1='63'

SW2	Meaning
'00'	No information given
'81'	Memory failure

Table 13 - Coding of SW2 when SW1='65'

SW2	Meaning
'00'	No information given
'81'	Logical channel not supported
'82'	Secure messaging not supported

Table 14 - Coding of SW2 when SW1='68'

SW2	Meaning
'00'	No information given
'81'	Command incompatible with file structure
'82'	Security status not satisfied
'83'	Authentication method blocked
'84'	Referenced data invalidated
'85'	Conditions of use not satisfied
'86'	Command not allowed (no current EF)
'87'	Expected SM data objects missing
'88'	SM data objects incorrect

Table 15 - Coding of SW2 when SW1='69'

SW2	Meaning
'00'	No information given
'80'	Incorrect parameters in the data field
'81'	Function not supported
'82'	File not found
'83'	Record not found
'84'	Not enough memory space in the file
'85'	Lc inconsistent with TLV structure
'86'	Incorrect parameters P1-P2
'87'	Lc inconsistent with P1-P2
'88'	Referenced data not found

Table 16 - Coding of SW2 when SW1='6A'

APDUs used by a POS to dialog with a MER

Introduction

Note about TLV: A TLV record is always defined with a Tag (2 bytes) prefixed by a qualifier “primitive” (0x5F) or “constructive” (0x7F), a Length coded with (2 bytes) and at least the Value with a size corresponding to the Length.

MER APDU Command

All charging MER commands are encapsulated in the body of an ISO 7816 command APDU (case 4) as explained below.

	Header (req)				Body		
Case 1: No cmd data & No resp required	CLA	INS	P1	P2			
Case 2: No cmd data & Yes resp required	CLA	INS	P1	P2	Le		
Case 3: Yes cmd data & No resp required	CLA	INS	P1	P2	LC	Data field	
Case 4: Yes cmd data & Yes resp required	CLA	INS	P1	P2	LC	MER API Cmd	Le

The MER Command (with parameters) is represented with a TLV format into a bytes array.

As the connection between the POS and the MER is not always secure, the MER Command is itself made of a security header and a command body that is ciphered.

MER API Command (0x0016)				
Security Header			MER Cmd Body TLV	
API_ORIGINATOR	SERVICE_PROVIDER_ID	SERVICE _PROVIDER_ TOKEN	TL	ciphered data val Byte[]

Field name	Description	Tag	Length (Bytes)	Values
API ORIGINATOR	The type of the API that is called, because each API uses different security policies and keys. From the POS device only the POS and READER APIs are available.	**0x2D**	1	**API_ORIGINATOR_SERVER = 0x00;** **API_ORIGINATOR_VIEWER = 0x01;** **API_ORIGINATOR_POS = 0x02** **API_ORIGINATOR_READER = 0x03;**
SERVICE_PROVIDER_ID	This field represents the hash of the SERVICE_PROVIDER_CODE produced by the hash () function defined above.	**0x1A**	2	(variable)
SERVICE_PROVIDER_TOKEN	It represents a signed value of a Service Provider Alea that is used to authenticate the POS owner. This token is generated by the ODC server and transmitted to the Service Provider previously for being installed into the POS.	**0x21**	8	(variable)
BODY TLV		**0x17**	variable	(variable)

MER APDU Response

The response returned by the MER is also a byte array stored into the body of the ISO 7816 APDU response as shown below.

Body	SW Trailer
MER (fragment) Response	SW1-SW2

The MER response fragment is ciphered and is structured as following:

MER (fragment) Response		
Data Length B1	Data Length B2	Ciphered Data Byte[]

This fragment can be re-iterated if the SW is set to 0x6310 (see below).

When the response of the MER exceeds 250 bytes, those fragments are not returned into the same APDU COMMAND-RESPONSE.

If the caller receives a ISO 7816 response with the Status Word set to 0x6310, the caller will have to send the INS_GET_RESPONSE (0xC0) C-APDU for getting the next fragment of the response until the SW's 0x9000 value.

In addition, sometimes when the ciphered data size (which corresponds to a complete entity MER object – ie an eToken containing long identifiers -) exceeds 250 bytes, an additional response buffer can be generated (a split of the ciphered data) by the MER (at low level) that won't be prefixed by B1||B2.

So, the best way to proceed in a POS for building the MER response is as following:

1. Allocate a final response byte array with a sufficient size (barray[8128] seems comfortable).
2. Send the MER instruction C-APDU
3. Read the response
4. If the response ends with 0x6310 value, append the response to the final response byte array by removing the 2 end bytes (0x6310) and then send the INS_GET_RESPONSE (0xC0) C-APDU and jump to the previous step.
5. If the response ends with the 0x9000 appends the response to the final response byte array.

At the end of the process, the POS will have built a response buffer that would be similar to the following barray:

MER fragment Response 1 (A&B)				...	MER fragment Response n			SW
Data 1 Length B1	Data 1 Length B2	Ciphered Data 1.A Byte[]	Ciphered Data 1.B Byte[]		Data n Length B1	Data n Length B2	Ciphered Data n Byte[]	0x9000

To decode the complete message the user will have to extract and de-cipher each data byte [] from the array above based on data length that will match each data fragment.

Then, by concatenating the decrypted byte arrays, the POS is able to reconstitute the complete response of the MER APDU initial command.

Error Status Word for POS MER API

Error name	Description	SW1 || SW2 Values
INVALID_MER_STATUS_IS_LOCKED	The MER is locked and cannot be used for charging operations.	**0x6BF4**
INVALID_MER_STATUS_NOT_YET_ACTIVATED	The MER applet has been installed but not yet personalized.	**0x6BF3**
ETOKEN_FULL_ARRAY_EXCEPTION	When too many eTokens (not yet flushed) are stored into the SE, the MER	**0x6B42**

	is no more able to perform the API calls (no enough memory) and returns systematically this SW.	
SW_CONDITIONS_NOT_SATISFIED	The merchant POS tries to charge a MER issued from an ODC platform that doesn't reference the merchant	**0x6985**
TAG_NOT_FOUND	A TLV is missing and the message is malformed	**0x7400**
SW_COMMAND_NOT_ALLOWED	The MER API is not called with the correct key	**0x6986**
AMOUNT_INVALID *(only with debit/credit)*	When the amount is not positive or out of bound.	**0x6B40**
PIN_REQUIRED	The client's MER asks for the PIN number before executing the operation (amount too high, max count of transaction reached, etc.)	**0x6B34**
REMAINING_DATA (*)	This SW indicates that the caller has to read additional response before stopping the communication with the MER	**0x6310**
OK_SATUS	The response has been fully transmitted and the MER cmd has been successfully executed.	**0x9000**

(*) When the caller receives this Status Word, he's invited to get the rest of the response by sending the INS_GET_RESPONSE (0xC0) instruction as defined by ISO 7816 in the above section.

Commented Debit() Example

In order to allow the user to program different POS devices, we propose 2 API levels for describing the interactions between the POS and the MER: a high level API (Java) and a low level API (APDU). Thanks to that we hope that the user will be able to implement ODC on a large scope of POS devices.

Description

A SAP Cola vendor wants implement a POS (vending machine) able to debit the default and shared account into a customer's MER. This SAP Cola vendor has previously received 2 security elements from the ODC server at provisioning time:

- A Vendor Charging symmetric Key (for ciphering/deciphering the MER Command / Response)
- A Vendor Token that authenticates the vendor

JAVA POS Lib description

This describes an implementation of a POS debit thanks to the ODC Java POS library (com.sap.odc.mer.api.pos)

```
        // we create the debit operation
    DebitAccount debitAccount;
    debitAccount = new DebitAccount();
        // we initialize the debit operation with the Vendor Charging Key (VcK)
    debitAccount.setKey("858D88C4F3D19CAE1E3B8882C9904345A638167391C5E1E6");
        // we set the MER API setID that will be called (this API must be compliant with the VcK)
    debitAccount.setOriginator(APIOriginator.pos);
        // we initialize the debit operation with the Vendor Token (for authentication)
    debitAccount.setServiceProviderToken(new ByteArray("21397F742FE39389"));
        // we ask the MER to store and return the generated tokens
    debitAccount.setProof(Proof.storedAndReturned);
        // we set the payee ID
    debitAccount.setPayeeId("500039251");
        // we set the amount to debit
    debitAccount.setAmount(new BigDecimal("0.6"));
        // we set the amount currency
    debitAccount.setCurrency("EUR");
        // we set the service provider code
```

```
    debitAccount.setServiceProviderCode("sapcola");
        // we set the transaction date
    debitAccount.setCurrentDate(new Date());
```

APDUs description

This part describes the APDUs that are directly exchanged between the POS and the MER (trough NFC by example). With this example the APDUs below have been generated by the Java ODC POS library.

```
Command => Debit account proof:3 serviceProvider:sapcola serviceProviderId:A7FB accountCode:null
accountId: amount:0.6 currency:EUR currentDate:Tue Jun 25 16:17:04 CEST 2013
```

Conventions/format:

```
                                ..................  MER API CMD V.......  ............
ISO APDU CODES  API CMD TL  ODC SECURITY HEADER  BODY TL  BODY V (CIPHERED MSG)


.................  MER API RESPONSE  ...................
 RESP LENGTH                CIPHERED RESPONSE                SW
```

Command APDU description

```
CLA||INS||P1||P2||LC||API_CMD.T||API_CMD.L|| ORI.T||
ORI.L||ORI.V||SP_ID.T||SP_ID.L||SP_ID.V||SP_TOK.T||SP_TOK.L_||SP_TOK.V || BODY.T||BODY.L||BODY.V

CAPDU =>

80||D8||00||00||67||7F16||0063||5F2D||0001||02||5F1A||0002||A7FB||5F21||0008||21397F742FE393895F170048
9FD26510508A6C19A178606FD6377F31DC6B0F7740E2DEA94EFA2C9D0AB997E31214BA1A6C1604D632CBC4EFD7A42E73641CB5
35047386C5590706C46DD072C5B9048B62607BC919
```

Note 1: The following result presents the full MER CMD before ciphering that the reason why the total Length (0x005C) differs from the total length of the ciphered message above (0x0063). This is also the reason why the MER command BODY (0x0041) Length differs from the body's size above (0x0048).

D8: represents the Instruction Debit()

```
offset:0x0000 (0) 7F16 005C (92) //API CMD TAG
        offset:0x0004 (4) 5F2D 0001 (1) value:02 //TAG Type / ORIGINATOR field (here 0x02 means POS)
        offset:0x0009 (9) 5F1A 0002 (2) value:A7FB //  SERVICE_PROVIDER ID (hash("sapcola"))
                offset:0x000F (15) 5F21 0008 (8) value:21397F742FE39389 //SERVICE PROVIDER TOKEN
                                                                          ByteArray("21397F742FE39389"))
        offset:0x001B (27) 5F17 0041 (65)
value:7FD8003D5F0900086536C36AEB889CFA5F1A0002A7FB5F260001035F280009353030303393235315F24000500000017
705F2500034555525F3300051AAF504440 // MER CMD Body
        offset:0x001F (31) 7FD8 003D (61) // DEBIT OPERATION (next data represents the Debit() params)
                offset:0x0023 (35) 5F09 0008 (8) value:6536C36AEB889CFA // ALEA(8) used for
                                                                          randomizing the ciphered CMD
                                                                            message (also used as seed
                                                                           for ciphering the response)
                offset:0x002F (47) 5F1A 0002 (2) value:A7FB //PROVIDER_ID
                offset:0x0035 (53) 5F26 0001 (1) value:03 //PROOF_POLICY (PROOF_STORED_AND_RETURNED)
                offset:0x003A (58) 5F28 0009 (9) value:353030303339323531 //PAYEE_ID
                offset:0x0047 (71) 5F24 0005 (5) value:0000001770 //AMOUNT to debit
                offset:0x0050 (80) 5F25 0003 (3) value:455552 //CURRENCY (here, "EUR" in ASCII)
                offset:0x0057 (87) 5F33 0005 (5) value:1AAF504440 //CURRENT_DATE (Tue Jun 25
                                                                          16:17:04 CEST 2013)
```

NOTE 2: Please, note that the order of the MER command parameters is very important and must be imperatively respected.

NOTE 3: Please, note that this example doesn't set the account code (no TAG_ACCOUNT_CODE is used), so in that case, the default account will be debited.

Remarks regarding the ODC integration with Mobiliser

If no default account is found, Mobiliser has defined the notion of shared account (between subscriber

accounts) and this account can replace the missing default account

Response APDU description

```
RAPDU <=

00F0E9A49FCEC1A9B866D6B093C5DF6593DB13DE37FD419F7CFED471716A8B8243DC8A0634D09EB5638C3FF06A3BBE5F58E529
F1472833DF109842EE2902633E44DEDE45B0F68EBEFD4ABB6CEFB044E1EF5F8A9A8B4FDBB0B0E8716ECD076D32F30D3734920B
2558A685D6A9B2B5A005572CB0A4E4A9C20ED7E4DCF66E095E7B29B859FC6F25B71C71C5156F1FDFE74B5A9D0FF3BC651151CF
71E1A771639F151CE4AC4A72AD35D665778A8907FCA01CF77815DFB4B8E6B9AA79565EB6E7A32F31D417C5BFD5C70312505465
309BAB475E947A49FC35BEA7C2B6AB4F0132FC23A184CD0BCE57642DFD07237D43E2D8AECD9B9000
```

After having deciphered the response returned by the MER (the message length should have changed following this task, see the above note 1) we can analyze the response.

```
         Status: 9000 Time: 2.178 // ISO Status Word SUCCESSFUL
offset:0x0000 (0) 7FD9 00E6 (230) // DEBIT ACCOUNT Result
         offset:0x0004 (4) 5F2F 0002 (2) value:0000 // Operation STATUS (0 means OK)
         offset:0x000A (10) 5F18 0002 (2) value:0000 // REJECTION CODE (0 means no code)
         offset:0x0010 (16) 7F27 00D6 (214) // Returned TOKEN
                  offset:0x0014 (20) 5F29 0009 (9) value:353030303437373034 // PAYER_ID (500047704)
                  offset:0x0021 (33) 5F28 0009 (9) value:353030303339323531 // PAYEE_ID (500039251)
                  offset:0x002E (46) 5F2D 0001 (1) value:01 // Token TYPE
                  offset:0x0033 (51) 5F32 0002 (2) value:0002 // TOKEN ID
                  offset:0x0039 (57) 5F0E 0005 (5) value:1AAF504440 // Transaction DATE
                  offset:0x0042 (66) 5F24 0005 (5) value:0000001770 // Token AMOUNT
                  offset:0x004B (75) 5F25 0003 (3) value:455552 // CURRENCY
                  offset:0x0052 (82) 5F43 000B (11) value:64656661756C745F737661 // ACCOUNT CODE
                  offset:0x0061 (97) 5F7A 0001 (1) value:00 // TRANSACTION TYPE
                  offset:0x0066 (102) 5F2A 0080 (128)
value:6BCD72864705DAD202DE7CD8116B72A5CBEF192BD1217ABBDC5EA5CEB09A39E0C00FB20BA98900648CB29CA87A5108EB
11342440366780CF8CC9580E644202B8998A3FFFEDEEC77459F7F79E27B8F900A834A538E2A2D3ADF42677B14CD19030BE3E47
5E9EE903F0F20937331C7E94763A21EB397B1FCD02BDA412F7E0747C49  // Token SIGNATURE (RSA by default)

Response <= DebitAccountMERResponse status=0 rejectionCode=0
EToken tokenType=1 tokenId=0002 date=Tue Jun 25 16:17:04 CEST 2013 amount=0.6 currency=EUR
chargeType=0 accountCode=default_sva payeeId=500039251 payerId=500047704
signature=6BCD72864705DAD202DE7CD8116B72A5CBEF192BD1217ABBDC5EA5CEB09A39E0C00FB20BA98900648CB29CA87A51
08EB11342440366780CF8CC9580E644202B8998A3FFFEDEEC77459F7F79E27B8F900A834A538E2A2D3ADF42677B14CD19030BE
3E475E9EE903F0F20937331C7E94763A21EB397B1FCD02BDA412F7E0747C49
```

MER ISO7816-5 Applet Identifier (AID)

The MER AID value is:

F0000000000901

MER Dates encoding

All dates are coded into the MER, on a 5-bytes buffer based on the following format:

```
 * YY     DDD       D  hh    mm    ss    reserved
 * [IIIIIII-I][IIIIIIII]-[III-IIIII]-[IIIIII-II][IIII-IIII]
 * 0       1       2       3       4
 * YY: the year (7 bits)
 * DDD : the day of the year (9bits)
 * D: the day of the week (3bits)
 * hh : the hour (5bits)
 * mm : the minute (6bits)
 * ss : the second (6bits)
 * reserved (4bits)
```

This format introduces some constraints and some restrictions:

- The Day of Week must correspond to the date (no calendar check is performed).
- The year must be comprised between 2000 and 2127.
- No time zone is considered.

Please find below an example of Java code for parsing/encoding 5 bytes buffer & java dates:

```
public static Date toDate(byte[] barray) {
    Date date = null;

    if ((barray != null) && (barray.length == DATE_LENGTH)) {
        boolean used = false;
        for (byte b : barray) {
            if (b != -1) {
                used = true;
                break;
            }
        }
        if (!used) {
            return null;
        }

        Calendar calendar = Calendar.getInstance();

        short year = (short) ((barray[0] & 0xFE) >> 1);
        short dayOfYear = (short) (((short) (barray[0] & 0x01) << 8) | (barray[1] & 0xFF));
        short dayOfWeek = (short) ((barray[(short) 2] & (short) 0xE0) >> (short) 5);
        short hour = (short) (barray[2] & 0x1F);
        short minute = (short) ((barray[3] & 0xFC) >> 2);
        short second = (short) (((short) (barray[3] & 0x03) << 4) | ((short) (barray[4] & 0xF0) >>
4));
        calendar.set(Calendar.YEAR, year + 2000);
        calendar.set(Calendar.DAY_OF_YEAR, dayOfYear + 1);
        calendar.set(Calendar.HOUR_OF_DAY, hour);
        calendar.set(Calendar.MINUTE, minute);
        calendar.set(Calendar.SECOND, second);
        calendar.set(Calendar.MILLISECOND, 0);
        assert calendar.get(Calendar.DAY_OF_WEEK) == (dayOfWeek + 1);
        date = calendar.getTime();
    }
    return date;
}

public static ByteArray toByteArray(Date date) {
    byte[] barray = new byte[5];

    if (date != null) {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        short year = (short) (calendar.get(Calendar.YEAR) % 100);
        short dayOfYear = (short) (calendar.get(Calendar.DAY_OF_YEAR) - 1);
        short dayOfWeek = (short) (calendar.get(Calendar.DAY_OF_WEEK) - 1);
        short hour = (short) calendar.get(Calendar.HOUR_OF_DAY);
        short minute = (short) calendar.get(Calendar.MINUTE);
        short second = (short) calendar.get(Calendar.SECOND);

        barray[0] = (byte) ((year << 1) | ((dayOfYear & 0x0100) >> 8));
        barray[1] = (byte) (dayOfYear);
        barray[2] = (byte) ((dayOfWeek << 5) | hour);
        barray[3] = (byte) ((minute << 2) | ((second >> 4) & 0x0003));
        barray[4] = (byte) (second << 4);
    } else {
        barray[0] = (byte) 0xFF;
        barray[1] = (byte) 0xFF;
        barray[2] = (byte) 0xFF;
        barray[3] = (byte) 0xFF;
        barray[4] = (byte) 0xFF;
    }

    return new ByteArray(barray);
}
```

MER number encoding

Here is the number format used by the MER

```
B4||B3||B2||B1||B0 with :
        * B4: number sign (0xFF negative / 0x00 positive)
        * B3||B2: integer part (from 0 to 65535)
        * B1||B0: decimal part (limited to 4 decimal digits -.9999 -)
```

Please find below an example of code that map a BigDecimal (which respects the MER Number restrictions) on a byte array as defined above.

```
 public static ByteArray toByteArray(BigDecimal bigDecimal) {
     byte[] barray = new byte[5];
     if (bigDecimal != null) {
        short intPart = bigDecimal.abs().shortValue();
        short decPart = bigDecimal.abs().subtract(new BigDecimal(intPart)).multiply(new
BigDecimal(10000)).shortValue();
        byte negative = bigDecimal.signum() < 0 ? (byte) 0xFF : (byte) 0x00;
        barray[0] = negative;
        barray[1] = (byte) ((intPart & (short) 0xFF00) >> (short) 8);
        barray[2] = (byte) (intPart & (short) 0x00FF);
        barray[3] = (byte) ((decPart & (short) 0xFF00) >> (short) 8);
        barray[4] = (byte) (decPart & (short) 0x00FF);
    }
    return new ByteArray(barray);
 }
```

Commented Charge() Example

Description

This generic example shows to the user (the POS' owner), how to build a Chargeable Item and how to send it to the MER from the POS for charging. This user has previously received 2 security elements from the ODC server at provisioning time:

- A Vendor Charging symmetric Key (for ciphering/deciphering the MER Command / Response).
- A Vendor Token that authenticates the user.

JAVA POS Lib description

This 1st part describes an implementation of a POS charge() thanks to the ODC Java POS library (com.sap.odc.mer.api.pos).

With this example we make the assumption that a related offer has been defined and instantiated into the ODC server and we assume that the mobile (MER) holder has subscribed to this offer.
At last, we suppose that the above offer is attached to a Chargeable Item Class previously defined (the structure that defines the service and the Chargeable Items that can be sent by the POS) in the ODC server.

```
        // we create the charge() operation
Date now = new Date();
Charge charge = new Charge();
        // we build the Chargeable Item (CI)
TLVChargeableItem chargeableItem = new TLVChargeableItem();
        // we set the CI name
chargeableItem.setName("ciName");
        // we add the different properties that will be carried by the CI
chargeableItem.addTLVProperty(new TLVProperty("propNumber", (byte)
TLVConstants.TYPE_NUMBER, new BigDecimal("3.13")));
chargeableItem.addTLVProperty(new TLVProperty("propString", (byte)
TLVConstants.TYPE_STRING, "string value"));
chargeableItem.addTLVProperty(new TLVProperty("propDate", (byte)
TLVConstants.TYPE_DATE, now));
        // we register the CI into the charge() operation  (only ONE CI per call to charge() operation)
charge.setChargeableItem(chargeableItem);
        // we set the current date
charge.setCurrentDate(now);
        // we set the consumption date
charge.setConsumptionDate(now);
        // we set the service code that will be hashed (by using above function) for having a service ID
charge.setServiceCode("callService");
// we set the service provider code that will be hashed (by using above function) for having a SP ID
```

```
charge.setServiceProviderCode("sapProvider");
        // we ask the MER to store and return the generated tokens
charge.setProof(Proof.storedAndReturned);
        // we initialize the debit operation with the Vendor Charging Key (VcK)
charge.setKey("858D88C4F3D19CAE1E3B8882C9904345A638167391C5E1E6");
// we set the MER API setID that will be called (this API must be compliant with the VcK)
charge.setOriginator(APIOriginator.pos);
        // we initialize the charge operation with the Vendor Token (for authentication)
charge.setServiceProviderToken(new ByteArray("AF202DA8B3F53916"));
```

APDUs description

This part describes the APDUs that are directly exchanged between the POS and the MER (trough NFC by example). With this example the APDUs below have been generated by the Java ODC POS library.

In opposite of the previous example, we directly propose here the non-ciphered version of the APDUs that are exchanged between the POS and the MER (we make the assumption that the message has been deciphered). That's the reason why we have modified a little, the conventions used for the notation.

Conventions/format:

```
                                  ................. MER API CMD V...
ISO APDU CODES  API CMD TL  ODC SECURITY HEADER  BODY TLV MSG


.................. MER API RESPONSE ............
 RESP LENGTH                RESPONSE             SW
(number in decimal)
```

Command APDU description

```
CLA||INS||P1||P2||LC||API_CMD.T||API_CMD.L|| ORI.T||ORI.L||ORI.V||SP_ID.T||SP_ID.L||
SP_ID.V||SP_TOK.T||SP_TOK.L_||SP_TOK.V || BODY.T||BODY.L||BODY.V

CAPDU =>

80||F0||00||00||B6||7F16||00AC||5F2D||0001||02||5F1A||0002||3A5B|5F21||0008||AF202DA8B3F53916||5F17||0
091||7FF0008D5F090008222726870EB063585F1A00023A5B5F3000025E6B5F3300051ABD512AD05F3100051ABD512AD05F260
001035F3F0001FF7F7100555F72000238817F7300145F76000249585F750001005F77000500000305147F73001B5F7600026A1
C5F750001015F77000C737472696E672076616C75657F7300145F76000225155F750001025F7700051ABD512AD0
```

F0: represents the instruction Charge().

```
Tue Jul 09 17:10:45 CEST 2013 (current date that has been used)
offset:0x0000 (0) 7F16 00AC (172) //API CMD TAG
        offset:0x0004 (4) 5F2D 0001 (1) value:02 //TAG Type / ORIGINATOR field (here 0x02 means POS)
        offset:0x0009 (9) 5F1A 0002 (2) value:3A5B //  SERVICE_PROVIDER ID (hash("sapProvider"))
        offset:0x000F (15) 5F21 0008 (8) value:AF202DA8B3F53916 //SERVICE PROVIDER TOKEN
                                                                  ByteArray("AF202DA8B3F53916"))
        offset:0x001B (27) 5F17 0091 (145) // MER CMD Body
value:7FF0008D5F090008222726870EB063585F1A00023A5B5F3000025E6B5F3300051ABD512AD05F3100051ABD512AD05F26
0001035F3F0001FF7F7100555F72000238817F7300145F76000249585F750001005F77000500000305147F73001B5F7600026A
1C5F750001015F77000C737472696E672076616C75657F7300145F76000225155F750001025F7700051ABD512AD0
        offset:0x001F (31) 7FF0 008D (141) // CHARGE OPERATION (next data are the charge() params)
                offset:0x0023 (35) 5F09 0008 (8) value: 222726870EB06358 // ALEA(8) used for
                                                                          randomizing the ciphered CMD
                                                                            message (also used as seed
                                                                           for ciphering the response)

                offset:0x002F (47) 5F1A 0002 (2) value:3A5B //PROVIDER_ID (hash("sapProvider"))
                offset:0x0035 (53) 5F30 0002 (2) value:5E6B // SERVICE_ID (hash("callService"))
                offset:0x003B (59) 5F33 0005 (5) value:1ABD512AD0//CURRENT_DATE
                offset:0x0044 (68) 5F31 0005 (5) value:1ABD512AD0//CONSUMPTION_DATE
                offset:0x004D (77) 5F26 0001 (1) value:03 //PROOF (value means stored&returned)
                offset:0x0052 (82) 5F3F 0001 (1) value:FF //BLANK CHARGE Flag: this means true
                offset:0x0057 (87) 7F71 0055 (85) //CHARGEABLE_ITEM constructed TLV element
                  offset:0x005B (91) 5F72 0002 (2) value:3881 // CHARGRABLE_ITEM_ID (hash("ciName"))
                  offset:0x0061 (97) 7F73 0014 (20) //PROPERTY 1
                    offset:0x0065 (101) 5F76 0002 (2) value:4958//PROPERTY_ID ((hash("propNumber"))
                    offset:0x006B (107) 5F75 0001 (1) value:00// PROPERTY_TYPE (number)
                    offset:0x0070 (112) 5F77 0005 (5) value:0000030514//PROPERTY_VALUE – 3.13
                  offset:0x0079 (121) 7F73 001B (27) //PROPERTY 2
```

```
                    offset:0x007D (125) 5F76 0002 (2) value:6A1C //PROPERTY_ID ((hash("propString”))
                    offset:0x0083 (131) 5F75 0001 (1) value:01 // PROPERTY_TYPE (string)
                     offset:0x0088 (136) 5F77 000C (12) value:737472696E672076616C7565//PROPERTY_VALUE
                                                                                        “string value”
                  offset:0x0098 (152) 7F73 0014 (20) //PROPERTY 3
                    offset:0x009C (156) 5F76 0002 (2) value:2515//PROPERTY_ID ((hash("propDate”))
                    offset:0x00A2 (162) 5F75 0001 (1) value:02// PROPERTY_TYPE (date)
                    offset:0x00A7 (167) 5F77 0005 (5) value:1ABD512AD0//PROPERTY_VALUE
```

Response APDU description

```
RAPDU <=

00667FF1003E7F61003A5F3E00021D385F3000025E6B5F3600051AB18A51E05F65000200017F78001B5F7A000200085F240005
00000423285F90000870726570616964319000

offset:0x0000 (0) 7FF1 003E (62)// INS_CHARGE_RESULT
        offset:0x0004 (4) 7F61 003A (58)// PURCHASE_ORDER
          offset:0x0008 (8) 5F3E 0002 (2) value:1D38 // OFFER_ID ((hash("myOffer”))
          offset:0x000E (14) 5F30 0002 (2) value:5E6B //SERVICE_ID
          offset:0x0014 (20) 5F36 0005 (5) value: 1ABD512AD0// CHARGING_DATE (CONSUMPTION_DATE)
          offset:0x001D (29) 5F65 0002 (2) value:0001 // ORDER_STATUS
          offset:0x0023 (35) 7F78 001B (27) // TRANSACTION
            offset:0x0027 (39) 5F7A 0002 (2) value:0008 // TRANSACTION_TYPE
            offset:0x002D (45) 5F24 0005 (5) value:0000042328 // TRANSACTION AMOUNT
            offset:0x0036 (54) 5F90 0008 (8) value:7072657061696431 // TRANSACTION ACCOUNT REFERENCE
```

List of TLV MER APDUs

(C/P): Constructive (0x7F) / Primitive (0x5F) tag

Remark: The following table indicates the position of the field in message. This order MUST be respected.

MER POS API Command C-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**TAG_API_COMMAND**	**0x16**	var	var	C
1	**TAG_TYPE**	**0x2D**	**0x0001**	0: ODC_SERVER 1:ODC_VIEWER 2:ODC_POS 3:ODC_READER	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_SERVICE_PROVIDER_TOKEN**	**0x21**	**0x0008**	var	P
4	**TAG_API_COMMAND_BODY**	**0x17**	var	var (ciphered)	P

Debit() C-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_DEBIT_ACCOUNT_OP**	**0xD8**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_PROOF**	**0x26**	**0x0001**	0: NOT_REQUIRED 1: RETURNED 2: STORED 3: S_AND_R	P
4	**TAG_PAYEE_ID**	**0x28**	var (max is *0x18*)	var	P
5	***TAG_ACCOUNT_ID (Optional)***	**0x4A**	**0x0002**	var	P
6	**TAG_AMOUNT**	**0x24**	**0x0005**	var (positive)	P

7	**TAG_CURRENCY**	**0x25**	**0x0003**	ISO 4217 3-letters + “PNT” + “XXX”	P
8	**TAG_CURRENT_DATE**	**0x33**	**0x0005**	var	P

Debit() R-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_DEBIT_ACCOUNT_RESULT**	**0xD9**	var	var	C
1	**TAG_STATUS**	**0x2F**	**0x0002**	**0:OK** **1:REJECTED**	P
2	**TAG_TOKEN_REJECTION_CODE**	**0x1A**	**0x0002**	**0:NO_CODE** **1:INSUFFICIENT_BALANCE**	P
3	**TAG_TOKEN**	**0x27**	var	var	C

TOKEN APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**TAG_TOKEN**	**0x27**	var	var	C
1	**TAG_PAYER_ID**	**0x29**	var (max is *0x18*)	var	P
2	**TAG_PAYEE_ID**	**0x28**	var (max is *0x18*)	var	P
3	**TAG_TYPE**	**0x2D**	**0x0001**	**0:CHARGE** **1:DEBIT_CREDIT** **2:P2P**	P
4	**TAG_TOKEN_ID**	**0x32**	**0x0002**	var	P
5	**TAG_DATE**	**0x0E**	**0x0005**	var	P
6	**TAG_AMOUNT**	**0x24**	**0x0005**	var (positive)	P
7	**TAG_CURRENCY**	**0x25**	**0x0003**	ISO 4217 3-letters + “PNT” + “XXX”	P
8	**TAG_ACCOUNT_CODE**	**0x43**	var (max is *0x18*)	var	P
9	**TAG_TRANSACTION_TYPE**	**0x7A**	**0x0001**	**0:DEBIT** **1:CREDIT**	P
10	**TAG_SIGNATURE**	**0x2A**	**0x0080 with RSA (default) or 0x0008 with DESmac**	var	P

Credit() C-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_CREDIT_ACCOUNT_OP**	**0xDA**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_PROOF**	**0x26**	**0x0001**	0: NOT_REQUIRED 1: RETURNED 2: STORED 3: S_AND_R	P
4	**TAG_PAYEE_ID**	**0x28**	var (max is *0x18*)	var	P
5	***TAG_ACCOUNT_ID (Optional)***	**0x4A**	**0x0002**	var	P
6	**TAG_AMOUNT**	**0x24**	**0x0005**	var (positive)	P

7	TAG_CURRENCY	0x25	0x0003	ISO 4217 3-letters + “PNT” + “XXX”	P
8	TAG_CURRENT_DATE	0x33	0x0005	var	P

Credit() R-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_CREDIT_ACCOUNT_RESULT	0xDB	var	var	C
1	TAG_STATUS	0x2F	0x0002	2:OK 3:REJECTED	P
2	TAG_TOKEN_REJECTION_CODE	0x1A	0x0002	0:NO_CODE	P
3	TAG_TOKEN	0x27	var	var	C

Charge() C-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_CHARGE_OP	0xF0	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P
3	TAG_SERVICE_ID	0x30	0x0002	var	P
4	TAG_CURRENT_DATE	0X33	0x0005	var	P
5	TAG_CONSUMPTION_DATE	0X31	0x0005	var	P
6	TAG_PROOF	0x26	0x0001	0: NOT_REQUIRED 1: RETURNED 2: STORED 3: S_AND_R	P
7	TAG_BLANK_CHARGE	0x26	0x0001	0xFF: true 0x00: false	
8	TAG_CHARGEABLE_ITEM	0x71	var	var	C

Charge() R-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_CHARGE_RESULT	0xF1	var	var	C
1	TAG_PURCHASE_ORDER	0x61	var	var	C
2	TAG_OFFER_ID	0x3E	0x0002	var	P
3	TAG_SERVICE_ID	0x30	0x0002	var	P
4	TAG_CHARGING_DATE	0x36	0X0005	var	P
5	TAG_ORDER_STATUS	0x65	0X0001	0:paid 1:free 2:rejected 3:done (blank charge)	P
6	TAG_TRANSACTION	0X78	0x0005	var	C

CHARGEABLE ITEM APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_CHARGEABLE_ITEM	0x71	var	var	C
1	TAG_ CHARGEABLE_ITEM_ID	0x72	0x0002	var	P
2	TAG_PROPERTY	0x73	var	var	C

CHARGEABLE ITEM PROPERTY APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_PROPERTY	0x73	var	var	C
1	TAG_PROPERTY_ID	0x76	0x0002	var	P
2	TAG_PROPERTY_TYPE	0x75	0x0001	0:num 1:string 2:date	P
3	TAG_PROPERTY_VALUE	0x77	0x0005 or 0x0018	var	P

TRANSACTION APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_TRANSACTION	0x78	var	var	C
1	TAG_TRANSACTION_TYPE	0x7A	0x0002	var	P
2	TAG_AMOUNT	0x24	0x0005	var	P
3	TAG_TRANSACTION_REFERENCE	0x90	0x0008	var	P

CheckMERPin() C-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_CHECK_PIN_OP	0xF6	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_CURRENT_DATE	0x33	0x0008	var	P
3	TAG_PIN	0x07	var	var	P

CheckMERPin() R-APDU

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_CHECK_PIN_RESULT	0xD9	var	var	C
1	TAG_STATUS	0x2F	0x0002	0:OK FF:REJECTED	P
2	TAG_PIN_REMAINING_TRIES(*)	0x06	0x0002	var	P

(*) Optional - set if the checking fails.

Security protocol between the POS and the MER

Introduction

2 generated secure data are required by the POS to interact with the POS in a secure way:

- A vendor 192 bits 3-DES charging key (called VcK)
- A vendor Token that authenticates the vendor.

Those data are issued for each vendor by the ODC server, when the vendor catalog is created.
This is the role of the vendor to handle those elements when they have been transmitted by the ODC server.

Those data are not stored in the ODC DB. However, they can be regenerated by the ODC server on demand. The cipher algorithm that is used by the POS and the MER for dialoguing is the "DESedeCBC-PKCS5". This symmetric algorithm has been chosen because it's available on 99% of Secure Elements and payment Terminal that are equipped with crypto processors.

Remarks regarding the ODC integration with Mobiliser

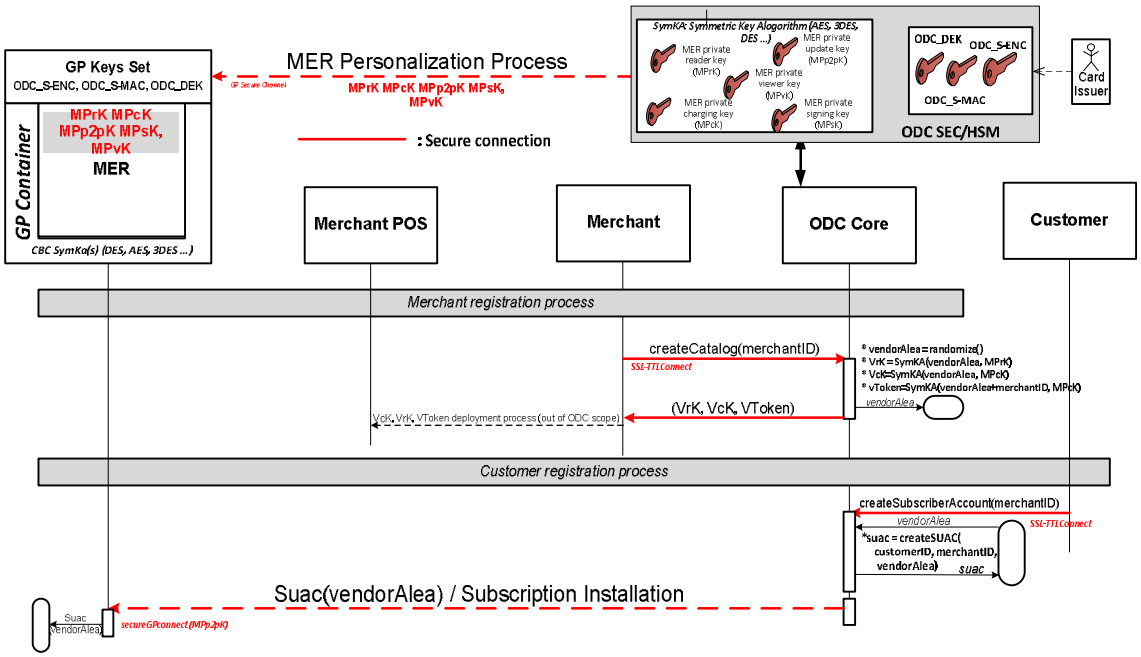
The 2 above secure data required for communicating with the POS (with an additional one, called Vendor Read Key used for usin,g the MER READ API) can be obtained from the ODC Web Service Contract provided by Money Mobiliser (see **`Odc.wdl`**) by calling the **`ReadMerchantKeys`** service**.**

Vendor security data generation

The vendors secure data is generated by the ODC sever from a vendor Alea and an ODC private key (MPcK: MER Private Charging Key).

The MPcK is created at ODC installation time. This key is installed into the MER during the MER personalization process.

The following sequence diagram presents the different steps performed by ODC.



Note 1: in addition to the VcK an additional key VrK, the Vendor Read Key, is also generated. It allows the vendor to access to the MER Read APIs for getting information about their accounts (Prepaid Account

balances, eTokens, counter values, etc.).

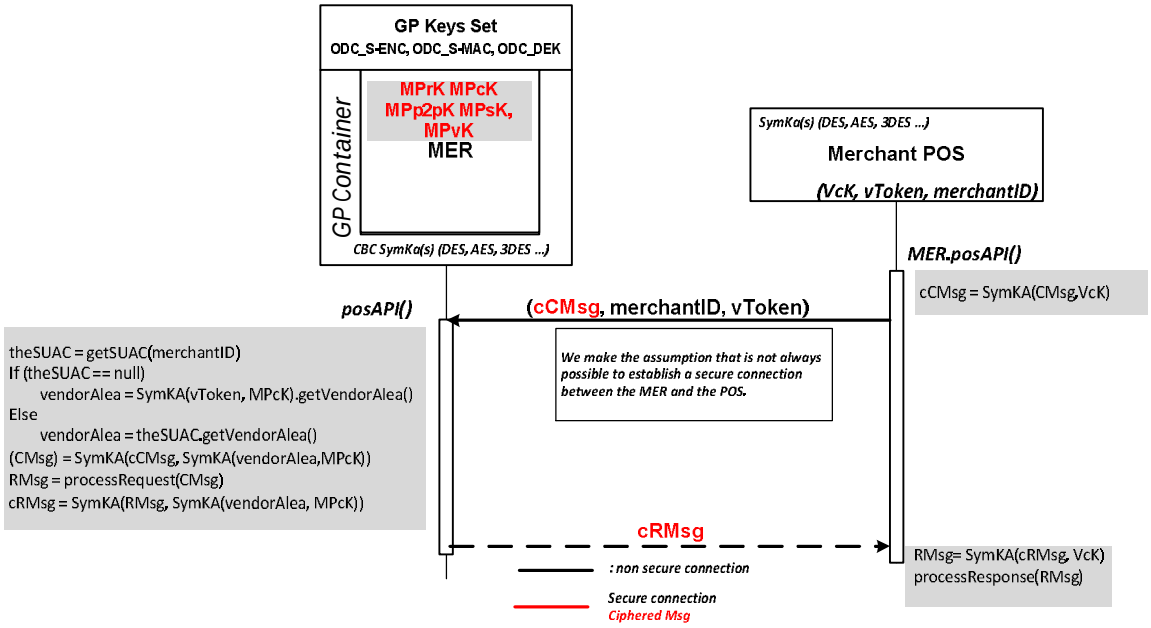
Note 2: the installation of the vendor secure data (VcK, VrK, vToken) in the POS is out of scope of this document.

Note 3: the merchant ID as mentioned on previous and next schemas is equivalent to the service provider ID as described in previous sections. Similarly, the Vendor/Merchant Token and the Service Provider Token represents the same object.

Note 4: the symmetric key encryption algorithm that is used for generating VcK, VrK and vToken is DESedeCBC-No padding (because ODC knows exactly the ALEA to cipher and wants to control the resulting key size – 192 bits).

Each time a end user will ask for having a commercial relationship with a vendor, a subscriber account (SUAC), will be created on the ODC server then synchronized with the MER. This subscriber account, that contains prepaid balances (SVA, Giftcards, …) contains also the vendor ALEA that be used by the MER to reconstruct the VCk and the VrK as explained in the next section.

How does it work?



Thanks to VcK the POS will cipher the MER C-APDU that must be performed by the MER.

By example, based on the previous example only the MER CMD Body value will have to be ciphered by the POS and the MER CMD length value and the API CMD Length will have to be adapted according to the new size of the ciphered message (part to be ciphered by the POS):

```
80||F0||00||00||B6||7F16||00AC||5F2D||0001||02||5F1A||0002||3A5B|5F21||0008||AF202DA8B3F53916||5F17||0
091||7FF0008D5F090008222726870EB063585F1A00023A5B5F3000025E6B5F3300051ABD512AD05F3100051ABD512AD05F260
001035F3F0001FF7F7100555F72000238817F7300145F76000249585F750001005F770005000000305147F73001B5F7600026A1
C5F750001015F77000C737472696E672076616C75657F7300145F76000225155F750001025F7700051ABD512AD0
```

F0: represents the instruction Charge().

```
Tue Jul 09 17:10:45 CEST 2013 (current date that has been used)
offset:0x0000 (0) 7F16 00AC (172) //API CMD TAG
        offset:0x0004 (4) 5F2D 0001 (1) value:02 //TAG Type / ORIGINATOR field (here 0x02 means POS)
        offset:0x0009 (9) 5F1A 0002 (2) value:3A5B // SERVICE_PROVIDER ID (hash("sapProvider"))
        offset:0x000F (15) 5F21 0008 (8) value:AF202DA8B3F53916 //SERVICE PROVIDER TOKEN (or vToken)
                                                    ByteArray("AF202DA8B3F53916"))
```

offset:0x001B (27) **5F17** 0091 (145) // **MER CMD Body**
value:7FF0008D5F090008222726870EB063585F1A00023A5B5F3000025E6B5F3300051ABD512AD05F3100051ABD512AD05F260001035F3F0001FF7F7100555F72000238817F7300145F76000249585F750001005F77000500000305147F73001B5F7600026A1C5F750001015F77000C737472696E672076616C75657F7300145F76000225155F750001025F7700051ABD512AD0

In addition, before ciphering the CMD body, the POS will have to prefix the CMD BODY value by adding a message ALEA (8 bytes) just after the API TAG.
This alea will guarantee the unicity of the ciphered message even if the same is called multiple times with the same argument (see the message's alea below)

CMD BODY value:
7FF0008D||5F09||0008||222726870EB06358||5F1A00023A5B5F3000025E6B5F3300051ABD512AD05F3100051ABD512AD05F260001035F3F0001FF7F7100555F72000238817F7300145F76000249585F750001005F77000500000305147F73001B5F7600026A1C5F750001015F77000C737472696E672076616C75657F7300145F76000225155F750001025F7700051ABD512AD0

offset:0x001F (31) **7FF0** 008D (141) // **CHARGE OPERATION (next data are the charge() params)**
offset:0x0023 (35) **5F09** 0008 (8) value: 222726870EB06358 // **ALEA(8) used for randomizing the ciphered CMD message (also used as seed for ciphering the response)**

3-DES Initialization Vector determination

When the POS will cipher the initial message with the 3-DES algorithm for the first time, a seed - the Initialization Vector (IV) - is required by the 3-DES. To share this seed between the MER and the POS, a 8 bytes random message is generated by the MER at each SELECT. Then the POS, from this SELECT response, will have to extract the IV from the message by keeping the 8 strong bytes.
This 8 bytes-message will be used as seed (3-DES IV) by the POS for ciphering all the APDU commands messages until the next SELECT APDU-command (until the NFC TAP for simplifying).

Example:

SELECT MER APDU command

00||A4||04||00||07F0000000000901:
0x00: class
0xA4: INS SELECT
0x04: P1
0x00: P2
0x07: LC
F0000000000901: MER AID

SELECT MER APDU response

Response:34EFAC4B9F365AB20490 00

The response returned by the SELECT MER instruction is always structured as following:
8-bytes random sequence||the MER status || the ISO 7816 SW.

The Initialization Vector (8 bytes) to use with 3-DES is the 8 first bytes returned by select: 34||EF||AC||4B||9F ||36||5A || B2

0x04 means that the applet is active and the 0x9000 result means that is OK.

For information, the available MER status values are:

```
MER_STATUS_VOID = (byte) 0x00; // ODC server side only
MER_STATUS_INSTALLED = (byte) 0x01;
MER_STATUS_PRE_PERSONALIZED = (byte) 0x02;
MER_STATUS_PERSONALIZED = (byte) 0x03;
MER_STATUS_ACTIVE = (byte) 0x04;
MER_STATUS_LOCKED = (byte) 0x05;
MER_STATUS_DISCARDED = (byte) 0x06;
MER_STATUS_CANCELED = (byte) 0x07; // ODC server side only
```

ODC java POS library

This guide presents an overview of the com.sap.odc.mer.api.pos package and doesn't replace the java documentation.

Those APIs are implemented as task that is registered into a macro container.
With each task/API, the user has to set the input parameters, to register the task into the macro container, to execute the macro and then read the output/result parameters from the API/task

Note: some restrictions must be applied on the following Java types:
- for String type, the size is limited to 24
- for BigDecimal, the value must be in between]-10,000, 10,000[

Charge() API

com.sap.odc.mer.api.pos.Charge	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
ServiceCode (String)	The code of the service (multiple services per merchant allowed)
ConsumptionDate (Date)	The consumption date can differ from the current date and the consumption date is used as reference date for triggering the recurring rates if these ones must be charged before charging the usage fees.
CurrentDate (Date)	Used for dating the transaction
Proof (enum)	Define the eToken policy (notRequired, returned, stored, storedAndReturned)
ChargeableItem (TLVChargeableItem)	Represents the usage consumption data record that must be rated.
Output/Result parameters	
PurchaseOrder (TLVPurchaseOrder)	The result of the charging process
Calculated values	
ServiceID (short)	Hash of the ServiceCode with the above hash() function
ServiceProviderID (short)	Hash of the ServiceProviderCode with the above hash() function

Debit() API

com.sap.odc.mer.api.pos.Debit	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
AccountCode (String)	The code of the account to debit
PayeeId (String)	The code that identifies the entity that will receive the debited amount.
Amount (BigDecimal)	The Amount to debit
Currency	The ISO-4217 3-letters code +"PNT" + "XXX" (XXX means no currency)
CurrentDate (Date)	Used for dating the transaction
Proof (enum)	(notRequired, returned, stored, storedAndReturned)
Output/Result parameters	
Status (Short)	OK or KO status
RejectionCode (short)	The reason why the debit has failed
Token (TLVToken)	The generated proof, when the transaction is accepted
Calculated values	
ServiceProviderID (short)	Hash of the ServiceProviderCode with the above hash() function

Credit() API

com.sap.odc.mer.api.pos.Credit	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
AccountCode (String)	The code of the account to debit
PayerId (String)	The code that identifies the entity who credits the amount.
Amount (BigDecimal)	The Amount to debit
Currency	The ISO-4217 3-letters code +"PNT" + "XXX" (XXX means no currency)
CurrentDate (Date)	Used for dating the transaction
Proof (enum)	(notRequired, returned, stored, storedAndReturned)
Output/Result parameters	
Status (Short)	OK or KO status
RejectionCode (short)	The reason why the debit has failed
Token (TLVToken)	The generated proof, when the transaction is accepted
Calculated values	
ServiceProviderID (short)	Hash of the ServiceProviderCode with the above hash() function

How to perform a call to the MER POS API from java code

In prerequisite, the caller has to implements at least the JAVA 1.6 JRE that contains the javax.smartcardio package.

The code example below show how to charge the MER through a ISO-7816 card reader (NFC or not)

```
        try {
            //factory able to connect to any compliant smartcard readers
            JDK6TerminalFactory jdk6TerminalFactory = new JDK6TerminalFactory();

            //commons GP and ISO-7816 api plus POS api mappings
            CardMappingList cardMappingList = new CardMappingList(new GPCardMapping(), new
APIPOSCardMapping());

            CommandTerminal commandTerminal = CommandTerminal(jdk6TerminalFactory.getTerminal("MY CARD
READER"), cardMappingList);

            // prepare macro
            Macro macro = new Macro();

            // atr task to establish a connection with the smartcard
            macro.addTask(new TaskATR());

            // select task to point to the MER applet
            TaskSelect taskSelect = new TaskSelect();
            // Unique and universal MER AID

            taskSelect.setAid(new ByteArray("F0000000000901"));
            macro.addTask(taskSelect);

            // prepare the Chargeable Item
            TLVChargeableItem chargeableItem = new TLVChargeableItem("buyCan");
            chargeableItem.addProperty(new TLVProperty("price", Property.TYPE_NUMBER, new
BigDecimal("1.0")));

            // charging operation
            Charge charge = new Charge();
            charge.setOriginator(APIOriginator.pos);
            charge.setServiceCode("drink");
            charge.setServiceProviderCode("sapcola");
            charge.setConsumptionDate(new Date());
            charge.setCurrentDate(new Date());
            charge.setChargeableItem(chargeableItem);
            charge.setProof(Proof.storedAndReturned);

            // register the API call in the macro
            macro.addTask(charge);

            //execute the macro
            macro.runMacro(commandTerminal);

            //get the result purchase order from the charge task.
            TLVPurchaseOrder purchaseOrder = charge.getPurchaseOrder();
```

```
        } catch (MacroErrorException e) {
            e.printStackTrace();
        } catch (CardErrorException e) {
            e.printStackTrace();
        }
```

SAP Cola full code example

Introduction

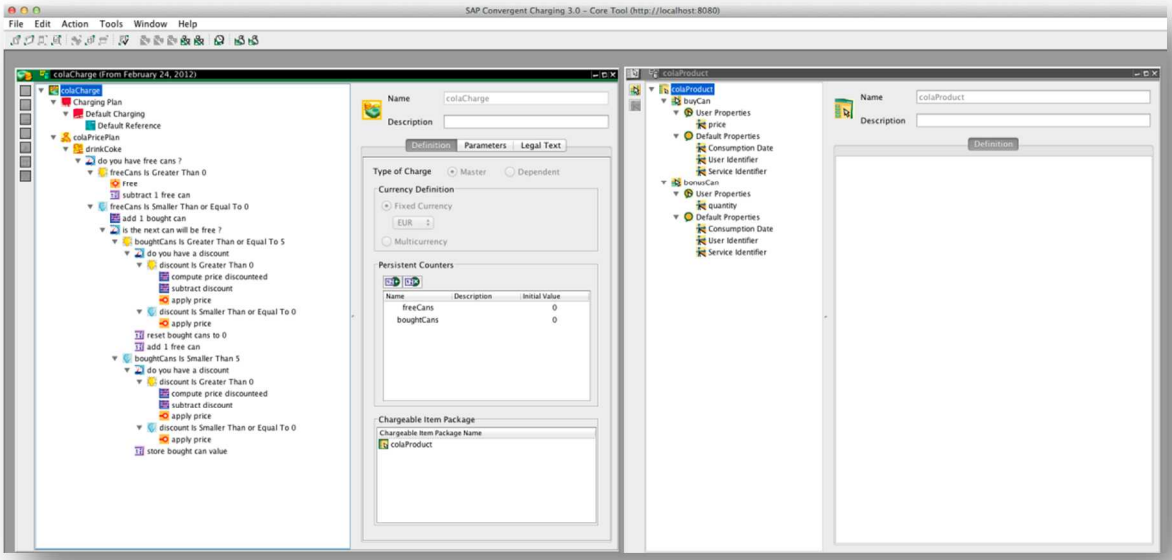
This example simulates a vending machine. The machine has been designed by using “Flash” and is interfaced with the code below through a “flashPlayer” class that will notify the PosCola class (that implements FlashPlayerListener interface) each time a button is pressed.

In addition, this program is connected to a NFC reader through the javax.smartcardio package.

Before describing the PosCola class, please note that a ChargeableItemPackage (ColaProduct) and a SAPcola offer (that contains the colaCharge) have been designed by the merchant SAPCola thanks to the graphical ODC Core Tool.

The SAPCola offer is subscribed by a end customer that owns the above smartphone.

The picture below presents these objects as they are defined with the tool.



PosCola.java

```
POSpackage com.sap.odc.pos.cola;

import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.util.Arrays;
import java.util.Date;
import java.util.Properties;

import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import chrriis.common.UIUtils;
import chrriis.dj.nativeswing.swtimpl.NativeInterface;
import chrriis.dj.nativeswing.swtimpl.components.FlashPlayerCommandEvent;
import chrriis.dj.nativeswing.swtimpl.components.FlashPlayerListener;
import chrriis.dj.nativeswing.swtimpl.components.JFlashPlayer;
import chrriis.dj.nativeswing.swtimpl.components.JWebBrowser;
import chrriis.dj.nativeswing.swtimpl.components.WebBrowserFunction;

import com.sap.odc.cload.CommandTerminal;
import com.sap.odc.cload.Macro;
import com.sap.odc.cload.MacroErrorException;
import com.sap.odc.commons.error.IError;
import com.sap.odc.commons.io.CardMappingList;
import com.sap.odc.commons.io.exception.CardErrorException;
import com.sap.odc.commons.io.exception.StatusWordErrorException;
import com.sap.odc.commons.util.ByteArray;
import com.sap.odc.gp.api.TaskATR;
import com.sap.odc.gp.api.TaskSelect;
import com.sap.odc.gp.api.intern.GPCardMapping;
import com.sap.odc.mer.api.core.APIOriginator;
import com.sap.odc.mer.api.core.MERApduHook;
import com.sap.odc.mer.api.pos.APIPOSCardMapping;
import com.sap.odc.mer.api.pos.DebitAccount;
import com.sap.odc.mer.api.pos.Proof;
import com.sap.odc.mer.models.ChargeableItem;
import com.sap.odc.mer.models.Property;
import com.sap.odc.mer.tlv.models.TLVEToken;
import com.sap.odc.pos.base.reader.ReadersSelectionListener;
import com.sap.odc.shared.TokenConstants;
import com.sap.odc.tool.cload.terminal.JDK6TerminalFactory;
```

```
public class PosCola extends JPanel implements FlashPlayerListener, ReadersSelectionListener {

    /**
     * Constructor of Example Frame
     */
    public PosCola() {
        super(new BorderLayout());

        properties = getPropertiesResource(System.getProperty("hd.config"));

        if (properties.getProperty("merchantName") == null) {
            JOptionPane.showMessageDialog(this, "The merchantName property is mandatory, Please check your config
file", "SAPCola", JOptionPane.ERROR_MESSAGE);
            System.exit(0);
            return;
        }
        if (properties.getProperty("merchantId") == null) {
            JOptionPane.showMessageDialog(this, "The merchantId property is mandatory, Please check your config file",
"SAPCola", JOptionPane.ERROR_MESSAGE);
            System.exit(0);
            return;
        }
        if (properties.getProperty("merchantToken") == null) {
            JOptionPane.showMessageDialog(this, "The merchantToken property is mandatory, Please check your config
file", "SAPCola",JOptionPane.ERROR_MESSAGE);
            System.exit(0);
            return;
        }
        if (properties.getProperty("merchantChargingKey") == null) {
            JOptionPane.showMessageDialog(this, "The merchantChargingKey property is mandatory, Please check your
config file", "SAPCola",JOptionPane.ERROR_MESSAGE);
            System.exit(0);
            return;
        }
        if (properties.getProperty("merchantCurrency") == null) {
            JOptionPane.showMessageDialog(this, "The merchantCurrency property is mandatory, Please check your config
file", "SAPCola",JOptionPane.ERROR_MESSAGE);
            System.exit(0);
            return;
        }

        flashPlayer = new JFlashPlayer();
        flashPlayer.load(getClass(), "/flash/SAP-Cola.swf");
        flashPlayer.addFlashPlayerListener(this);

        flashPlayer.getWebBrowser().registerFunction(new WebBrowserFunction("buyCan") {
            @Override
            public Object invoke(JWebBrowser webBrowser, Object... args) {
                LOG.debug("buyCan");
                return buyCan((String) args[0]);
            }
        });
        flashPlayer.getWebBrowser().registerFunction(new WebBrowserFunction("getMessage") {
            @Override
            public Object invoke(JWebBrowser webBrowser, Object... args) {
                LOG.debug("getMessage");
                return getMessage();
            }
        });

        add(flashPlayer, BorderLayout.CENTER);
    }

    public boolean buyCan(String type) {
        LOG.debug("buyCan {} ", type);
        BigDecimal price = null;
        if (type.equals("zero")) {
            price = new BigDecimal("0.7");
        } else if (type.equals("light")) {
            price = new BigDecimal("0.6");
        } else if (type.equals("original")) {
            price = new BigDecimal("1");
        } else if (type.equals("black")) {
            price = new BigDecimal("0.4");
        } else {
            price = new BigDecimal("0.5");

        }

        CommandTerminal commandTerminal;
        try {
            commandTerminal = waitForTerminal();
            if (commandTerminal == null) {
                return false;
            }
            sb = new StringBuilder();
            Macro macro = initMacro("F0000000000901", commandTerminal);

            DebitAccount debitAccount;
            debitAccount = new DebitAccount();
            debitAccount.setKey(properties.getProperty("merchantChargingKey"));
            debitAccount.setOriginator(APIOriginator.pos);
            debitAccount.setServiceProviderToken(new ByteArray(properties.getProperty("merchantToken")));
            debitAccount.setProof(Proof.storedAndReturned);
            debitAccount.setPayeeId(properties.getProperty("merchantId"));
```

```
            debitAccount.setAmount(price);
            debitAccount.setCurrency(properties.getProperty("merchantCurrency"));
            debitAccount.setServiceProviderCode(properties.getProperty("merchantName"));
            debitAccount.setCurrentDate(new Date());

            macro.addTask(debitAccount);

            MERApduHook hook = new MERApduHook();
            commandTerminal.addCommandHook(hook);
            commandTerminal.addResponseHook(hook);

            try {
                macro.runMacro(commandTerminal);
                if (debitAccount.getStatus() == TokenConstants.STATUS_DEBIT_OK) {
                    TLVEToken token = debitAccount.getToken();
                    sb.append("Status : PAID").append("\n");
                    sb.append("Amount : " + token.getAmount()).append("\n");
                    return true;
                } else {
                    sb.append("Status : ERROR").append("\n");
                    return false;
                }
            } catch (MacroErrorException exception) {
                LOG.error("MacroErrorException", exception);
                Throwable throwable = exception;
                do {
                    throwable = throwable.getCause();
                } while (throwable.getCause() != null);
                if (throwable instanceof StatusWordErrorException) {
                    IError error = ((StatusWordErrorException) throwable).getError();
                    sb.append(error.getMessage());
                } else {
                    sb.append(exception.getError().getMessage());
                }
            } catch (CardErrorException exception) {
                LOG.error("CardErrorException", exception);
                sb.append(exception.getError().getMessage());
            }

        } catch (InterruptedException exception) {
            LOG.error("InterruptedException", exception);
            sb.append(exception.getMessage());
        }

        return false;
    }

    public CommandTerminal waitForTerminal() throws InterruptedException {
        LOG.info("Approach your mobile phone...");

        JDK6TerminalFactory jdk6TerminalFactory = new JDK6TerminalFactory();
        int maxTimeout = 100;
        while (jdk6TerminalFactory.getAvailableTerminals().isEmpty() && maxTimeout > 0) {
            Thread.sleep(100);
            maxTimeout--;
        }

        if (!jdk6TerminalFactory.getAvailableTerminals().isEmpty()) {
            LOG.info("Mobile phone decteted on reader %s", jdk6TerminalFactory.getAvailableTerminalNames().get(0));
            return new CommandTerminal(jdk6TerminalFactory.getAvailableTerminals().get(0), cardMappingList);
        } else {
            LOG.info("No Mobile phone found in time");
            sb = new StringBuilder("No Mobile phone found in time");
            return null;
        }
    }

    protected ChargeableItem createPayChargeableItem(BigDecimal amount, String currency, BigDecimal type, String
merchantID) {
        ChargeableItem chargeableItem = new ChargeableItem();
        chargeableItem.setName("pay");
        chargeableItem.addProperty(new Property("amount", Property.TYPE_NUMBER, amount));
        chargeableItem.addProperty(new Property("currency", Property.TYPE_STRING, currency));
        chargeableItem.addProperty(new Property("type", Property.TYPE_NUMBER, type));
        chargeableItem.addProperty(new Property("merchantId", Property.TYPE_STRING, merchantID));

        return chargeableItem;
    }

    public String getMessage() {
        LOG.debug("getMessage {}", sb);
        return sb.toString();
    }

    /**
     * Called from Flash by ExternalInterface.call to indicate mouse over and out events on a Flash symbol.
     */
    public void notifyFlashMouseEvent(String event) {
        LOG.debug("Flash mouse event: " + event);
    }

    /**
     * Called if no flash is present then exit application
     */
    private void exitError(String message) {
```

```
        JOptionPane.showMessageDialog(null, message, "Error", JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    }

    @Override
    public void notifyReaderSelected(String reader) {
        // selectedReader = reader;
    }

    @Override
    public void commandReceived(FlashPlayerCommandEvent flashPlayerCommandEvent) {
        LOG.debug("commandReceived {} ", flashPlayerCommandEvent.getCommand());
        LOG.debug("commandReceived args {} ", Arrays.toString(flashPlayerCommandEvent.getParameters()));
        if (flashPlayerCommandEvent.getCommand().equals("quit")) {
            System.exit(0);
        }
    }

    @Override
    public String getIcon() {
        return "img/cola.jpg";
    }

    @Override
    public String getTitle() {
        return "SAP Cola";
    }

    public JFlashPlayer getFlashPlayer() {
        return flashPlayer;
    }

    public static void main(String[] args) {
        UIUtils.setPreferredLookAndFeel();
        NativeInterface.open();
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                PosCola posCola = new PosCola();
                JFrame frame = new JFrame(posCola.getTitle());
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setContentPane(posCola);
                frame.setIconImage(Toolkit.getDefaultToolkit().getImage(ClassLoader.getSystemResource(
posCola.getIcon())));
                frame.setSize(1024, 768);
                frame.setLocationByPlatform(true);
                frame.setVisible(true);
            }
        });
        NativeInterface.runEventPump();
    }

    private static Properties getPropertiesResource(String configFile) {
        LOG.debug("load properties from {}", configFile);
        Properties propertiesResource = new Properties();
        try {
            propertiesResource.load(new FileInputStream(configFile));
        } catch (IOException exception) {
            LOG.error("Unable to load properties ", exception);
        }

        return propertiesResource;
    }

    private static Macro initMacro(String aid, CommandTerminal commandTerminal) {
        Macro macro = new Macro();
        macro.addTask(new TaskATR());

        TaskSelect taskSelect = new TaskSelect();
        taskSelect.setAid(new ByteArray(aid));
        macro.addTask(taskSelect);

        return macro;
    }


    private Properties properties;
    private JFlashPlayer flashPlayer;
    private StringBuilder sb = new StringBuilder();
    private static final long serialVersionUID = 1L;
    private static final Logger LOG = LoggerFactory.getLogger(PosCola.class);

    protected static final CardMappingList cardMappingList = new CardMappingList(new GPCardMapping(), new
APIPOSCardMapping());

}
```

An extension to the POS APIs: the READ API

This API allows a merchant/vendor to read his own data that are stored into a MER installed on customer's device.

The working of this API is completely similar to the POS API. However, this API requires a dedicated vendor key that differs from the vendor charging key: This key is called Vendor read Key (VrK).

The read API limits the access to the data that belongs to the vendor subscriber account and the accessible data are the tokens, merchant's Prepaid Account, merchant subscriptions data (parameters and related counters), charge history and refill history.

Please find below the description of this API that proposes a high (Java) and low (bytes) interfaces.
We recommend using the Java interface that is easier to manipulate but if the merchant needs to access to customer's data from a basic terminal; the low level API description will be useful, the high level interface requiring a JVM for running.

SearchMERChargingHistory()

This API allows the merchant to access to the customer's charging history limited to the list of purchase orders generated by the subscriptions issued from merchant's offers.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERChargingHistory	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
PurchaseOrders (List<TLVPurchaseOrder>)	Return the list of purchase orders stored into the history

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_CHARGING_HISTORY_OP**	**0xCA**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
0	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_CHARGING_HISTORY_RESULT**	**0xCB**	var	var	C
1	**TAG_PURCHASE_ORDER (*) (sequence of)**	**0x61**	**var**	var	C

(*)see purchase order in previous document section

SearchMERCreditLimit()

This API allows a merchant to read the credit limits of his customer which has been set in the related subscriber account (max 3).

Remarks regarding the ODC integration with Mobiliser

This API is not used because Mobiliser doesn't allow creating credit limits.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERCreditLimit	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
CreditLimits (List<TLVCreditLimit>)	Return the list of credit limits that are set on the related merchant account.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_CREDIT_LIMIT_OP**	**0xCF**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_CREDIT_LIMIT_RESULT	0xD0	var	var	C
1	TAG_CREDIT_LIMIT (sequence of)	0x42	var	var	C

TAG CREDIT_LIMIT

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_CREDIT_LIMIT	0x42	var	var	C
1	TAG_ACCOUNT_ID	0x4A	0x0002	var	P
2	*TAG_ACCOUNT_CODE*	0x43	var	var	P
3	TAG_AMOUNT	0x24	0x0005	var (positive)	P
4	TAG_CURRENCY	0x25	0x0003	ISO 4217 3-letters + “PNT” + “XXX”	P
5	TAG_ACCOUNT_RECURRING_TYPE	0x4B	0x0002	0: Daily 1:Weekly 2:Monthly 3:Yearly -5:Unset -1:Not Recurring	P
6	TAG_ACCOUNT_RECURRING_DATE	0x4C	0x0005	var	P
7	TAG_ACCOUNT_RECURRING_AMOUNT	0x4D	0x0005	var	P

SearchMERETokens()

Allows the merchant to read all the Electronic Tokens (currently registered into the MER) for which the merchant is involved as payee or payer.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERETokens	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
ETOkens (List<TLVEToken>)	the list of ETokens

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_TOKENS_OP	0xC3	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_TOKENS_RESULT	0xC4	var	var	C
1	TAG_TOKEN(*) (sequence of)	0x27	var	var	C

(*) See TLV description in previous POS API section.

SearchMERExternalAccount()

Returns the list of the external account defined in the merchant's subscriber account.

Remarks regarding the ODC integration with Mobiliser

This API is not used because Mobiliser doesn't create/use some external accounts.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERExternalAccount	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
ExternalAccounts (List< TLVExternalAccount>)	the list of External accounts defined into the merchant account.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_EXTERNAL_ACCOUNT_OP	0xCD	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_EXTERNAL_ACCOUNT_RESULT	0xCE	var	var	C
1	TAG_EXTERNAL_ACCOUNT (sequence of)	0x41	var	var	C

TAG EXTERNAL_ACCOUNT

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_EXTERNAL_ACCOUNT	0x41	var	var	C
1	TAG_ACCOUNT_ID	0x4A	0x0002	var	P
2	*TAG_ACCOUNT_CODE*	0x43	var	var	P
3	*TAG_ACCOUNT_OWNER*	0x44	var	var	P
4	TAG_AMOUNT	0x24	0x0005	var (positive)	P
5	TAG_CURRENCY	0x25	0x0003	ISO 4217 3-letters + "PNT" + "XXX"	P

SearchMERPrepaidAccount()

Returns the list of Prepaid Accounts which are defined by the merchant in a subscriber account.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERPrepaidAccount	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	

PrepaidAccounts (List<TLVPrepaidAccount>)	The list of Prepaid Accounts defined into the merchant MER's account.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_PREPAID_ACCOUNT_OP	0xB2	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_PREPAID_ACCOUNT_RESULT	0xB3	var	var	C
1	TAG_PREPAID_ACCOUNT (sequence of)	0x40	var	var	C

TAG PREPAID_ACCOUNT

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_PREPAID_ACCOUNT	0x40	var	var	C
1	TAG_ACCOUNT_ID	0x4A	0x0002	var	P
2	*TAG_ACCOUNT_CODE*	0x43	var	var	P
3	*TAG_ACCOUNT_OWNER*	0x44	var	var	P
4	TAG_AMOUNT	0x24	0x0005	var (positive)	P
5	TAG_CURRENCY	0x25	0x0003	ISO 4217 3-letters + "PNT" + "XXX"	P
6	TAG_ACCOUNT_EMPTY_LIMIT	0x48	0x0005	var	P
7	TAG_ACCOUNT_BLOCKED_STATE_DATE	0x45	0x0005	var	P
8	TAG_ACCOUNT_LOCKED_STATE_DATE	0x46	0x0005	var	P
9	TAG_ACCOUNT_CLOSED_STATE_DATE	0x47	0x0005	var	P

SearchMERRefillHistory()

This API allows the merchant to access to the customer's refill history which has been performed by the customer on Prepaid Accounts defined into the merchant's MER account (subscriber account).

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERRefillHistory	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
RefillOrders (List<TLVRefillOrder>)	The list of refill orders performed into the merchant MER's account.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_REFILL_HISTORY_OP	0xB4	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_REFILL_HISTORY_RESULT	0xB5	var	var	C
1	TAG_REFILL_ORDER (sequence of)	0x68	var	var	C

TAG REFILL_ORDER

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_REFILL_ORDER	0x68	var	var	C
1	*TAG_ACCOUNT_CODE*	0x43	var	var	P
2	TAG_REFILL_ORDER_AMOUNT	0x69	0x0005	var	P
3	*TAG_REFILL_ORDER_PRE_AMOUNT*	0x6A	0x0005	var	P
4	TAG_REFILL_ORDER_POST_AMOUNT	0x6B	0x0005	var	P
5	TAG_REFILL_ORDER_STATUS	0x6C	0x0002	0: refilled 1: rejected	P
6	TAG_REFILL_ORDER_REJECT_CODE	0x6D	0x0002	0:balance rejected 1: invalid account code	P
7	TAG_REFILL_DATE	0x35	0x0005	var	P

SearchMERSubscriberAccount()

This API allows the merchant to access to his subscriber code (if it exists) that will contain all the nested objects such as Eternal Accounts, Credit Limits and Prepaid Accounts.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERSubscriberAccount	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
TLVSubscriberAccount	The merchant MER's account.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_SUBSCRIBER_ACCOUNT_OP	0xB0	var	var	C
1	TAG_ALEA	0x09	0x0008	var	P
2	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	INS_SEARCH_MER_SUBSCRIBER_ACCOUNT_RESULT	0xB1	var	var	C
1	TAG_SUAC	0x10	var	var	C

TAG SUBSCRIBER_ACCOUNT

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_SUAC	0x10	var	var	C
1	TAG_SUAC_ID	0x11	0x0002	var	P
2	TAG_SUAC_CODE	0x14	var	var	P
3	TAG_SUAC_DEFAULT_CHARGE_ACCOUNT_ID	0x12	0x0002	var	P
4	TAG_SUAC_SUBSCRIBER	0x13	0x0005	var	P
5	TAG_SERVICE_PROVIDER_CODE	0x20	var	var	P
6	TAG_SERVICE_PROVIDER_ID	0x1A	0x0002	var	P

SearchMERSubOfferCodes

This API allows the merchant to access to the list of subscribed offer codes. It returns the Offer codes stored into the embedded subscription installed into the end-user Secure Element.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERSubOfferCodes	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
Output/Result parameters	
Codes (List<String>)	The list of the subscribed offer codes.

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_SUB_OFFER_CODES_OP**	**0xB6**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_SUB_OFFER_CODES_RESULT**	**0xB7**	var	var	C
1	**TAG_OFFER_CODE (optional) (sequence of)**	**0x22**	var	var	C

SearchMERSubscriptionInfo

Remarks regarding the ODC integration with Mobiliser

Each API that returns a subscription (**TLVSubscription**) will reference a simplified version of the ODC subscription. This simplified version of the ODC **Subscription** can be consulted in the WS definition file (Odc.wsdl) and is named "**SubscriptionDD3Form**". The "**SubscriptionDD3Form**" can be viewed as a TLVSubscription with restrictions. Here is an overview of the SubscriptionDD3Form (for the complete definition, please consult the Odc.wsdl file):

```
    <xs:complexType name="SubscriptionDD3Form">
        <xs:sequence>
            <xs:element ref="parameterSet" minOccurs="0" />
            <xs:element ref="counterSet" minOccurs="0" />
            <xs:element ref="translationInstanceSet" minOccurs="0" />
            <xs:element name="chargeActivationD3Form" type="ChargeActivationD3Form" minOccurs="1"
maxOccurs="10" />
        </xs:sequence>
        <xs:attribute name="code" type="core:String24" use="required" />
        <xs:attribute name="merchantId" type="base:idLong" use="optional" />
        <xs:attribute name="customerId" type="base:idLong" use="optional" />
        <xs:attribute name="offerCode" type="xs:string" use="required" />
        <xs:attribute name="effectiveDate" type="xs:dateTime" />
        <xs:attribute name="expirationDate" type="xs:dateTime" />
        <xs:attribute name="dependentMaxDeph" type="xs:short" fixed="3" />
    </xs:complexType>
==========================================================================================
    <xs:complexType name="ChargeActivationD3Form">
        <xs:sequence>
            <xs:element ref="parameterSet" minOccurs="0" maxOccurs="1" />
            <xs:element ref="counterSet" minOccurs="0" maxOccurs="1" />
            <xs:element ref="translationInstanceSet" minOccurs="0" maxOccurs="1" />
            <xs:element name="serviceId" type="core:String24" minOccurs="0" maxOccurs="1">
                <xs:annotation>
                    <xs:documentation>
                        Mandatory if the charge type is master.
```

```
                </xs:documentation>
            </xs:annotation>
        </xs:element>
        <xs:element ref="chargingMapping" minOccurs="0" maxOccurs="unbounded">
            <xs:annotation>
                <xs:documentation>
                    If not set, the charging is done on the default account.
                </xs:documentation>
            </xs:annotation>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="chargeConditionOccurenceNumber" type="xs:short" default="0">
          <xs:annotation>
        <xs:documentation>
            If several Charge Activations reference the same Charge Condition, each CA
                                will be indexed by a number indicating the CA range.
        </xs:documentation>
      </xs:annotation>
          </xs:attribute>
    <xs:attribute name="description" type="xs:string" use="optional" />
    <xs:attribute name="effectiveDate" type="xs:dateTime" use="required" />
    <xs:attribute name="expirationDate" type="xs:dateTime" />
    <xs:attribute name="chargeConditionCode" type="xs:string" use="required" />
    <xs:attribute name="chargeType" type="CAType" default="master" />
    <xs:attribute name="dependentMaxCount" type="xs:integer" fixed="3" />
          <xs:attribute name="internalCode" type="core:String24">
                <xs:annotation>
        <xs:documentation>
            The internalCode is calculated by ODC.
        </xs:documentation>
      </xs:annotation>
          </xs:attribute>
</xs:complexType>
```

This API allows the merchant to access to the information of a subscription identified by its offer code.

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERSubscriptionInfo	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
OfferCode (String)	The code of the offer
Output/Result parameters	
Subscription (TLVSubscription)	The details of the subscription that will contain the data for the following fields: - subscription code - subscription ID - the service provider code - the offer code - the offer ID - the effective date of the subscription - the expiration date of the subscription

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_INFO_OP**	**0xBE**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_OFFER_ID**	**0x3E**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_INFO_RESULT**	**0xBF**	var	var	C

1	TAG_SUBSCRIPTION	0x51	var	var	C

TAG SUBSCRIPTION

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_SUBSCRIPTION	0x51	var	var	C
1	TAG_SUBSCRIPTION_CODE	0x52	var	var	P
2	TAG_SUBSCRIPTION_ID	0x56	0x0002	var	P
3	TAG_SERVICE_PROVIDER_CODE	0x20	var	var	P
4	TAG_OFFER_ID	0x3E	0x0002	var	P
5	TAG_OFFER_CODE	0x22	var	var	P
6	TAG_CHARGE_ACTIVATION (*) (sequence of)	0x1A	var	var	C
7	TAG_COUNTER (sequence of)	0x7D	var	var	C
8	TAG_PARAMETER (sequence of)	0x7C	var	var	C

(*) represents the list of master charge activation

TAG CHARGE_ACTIVATION

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_CHARGE_ACTIVATION	0x1A	var	var	C
1	TAG_CHARGE_ACTIVATION_CODE	0x5C	var	var	P
2	TAG_CHARGE_ACTIVATION_ID	0x5D	0x0002	var	P
3	TAG_CHARGE_ACTIVATION_EFFECTIVE_DATE	0x60	0x0005	var	P
4	TAG_CHARGE_ACTIVATION_EXPIRATION_DATE	0x5F	0x0005	var	P
5	TAG_CHARGE_ACTIVATION_CANCELLATION_DATE	0x5E	0x0005	var	P
6	TAG_CHARGE_ACTIVATION_ONE_SHOT (sequence of)	0x62	0x0001	0: applied other not applied	P
7	TAG_CHARGE_ACTIVATION_RECURRING_DATE (sequence of)	0x61	0x0001	0: applied other not applied	P
8	TAG_COUNTER (sequence of)	0x7D	var	var	C
9	TAG_PARAMETER (sequence of)	0x7C	var	var	C
10	TAG_CHARGE_ACTIVATION (*) (sequence of)	0x1A	var	var	C

(*) this nested structure represents the list of dependent charge activation associated to the current master master charge.

TAG COUNTER

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	TAG_COUNTER	0x7D	var	var	C
1	TAG_NAME	0x2C	var	var	P
2	TAG_VALUE	0x2B	0x0005	var	P

TAG PARAMETER

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**TAG_PARAMETER**	**0x7C**	**var**	var	C
1	**TAG_NAME**	**0x2C**	**var**	var	P
2	**TAG_TYPE**	**0x2D**	**0x0001**	**0:number** **1:string** **2:date**	P
3	**TAG_VALUE**	**0x2B**	**var**	var	P

SearchMERSubscriptionCounters

This API allows the merchant to access to the counters' values contained in a subscription identified by its offer code.

Remarks regarding the ODC integration with Mobiliser

Each API that returns a subscription (**TLVSubscription**) references a simpler version of the ODC subscription. This simplified version of the ODC **Subscription** can be consulted in the WS definition file (Odc.wsdl) and is renamed "**SubscriptionDD3Form**". The "**SubscriptionDD3Form**" can be viewed as a TLVSubscription with restrictions. Here is an overview of the SubscriptionDD3Form (for the complete definition, please consult the Odc.wsdl file):

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERSubscriptionCounters	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
OfferCode (String)	The code of the offer
Output/Result parameters	
Subscription (TLVSubscription)	The details of the subscription that will contain the data for the following fields which contain counters (List<TLVCounter>): - SharedCounters (List<TLVCounter> - ChargeActivations (List<TLVChargeActivation>) o Counters (List<TLVCounter>

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_COUNTER_OP**	**0xB8**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_OFFER_ID**	**0x3E**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_COUNTER_RESULT**	**0xB9**	var	var	C
1	**TAG_SUBSCRIPTION (*)**	**0x51**	**var**	var	C

(*) defined in previous section

SearchMERSubscriptionParameters

This API allows the merchant to access to the parameters' values contained in a subscription identified by its offer code.

Remarks regarding the ODC integration with Mobiliser

Each API that returns a subscription (**TLVSubscription**) references a simpler version of the ODC subscription. This simplified version of the ODC **Subscription** can be consulted in the WS definition file (Odc.wsdl) and is renamed "**SubscriptionDD3Form**". The "**SubscriptionDD3Form**" can be viewed as a TLVSubscription with restrictions. Here is an overview of the SubscriptionDD3Form (for the complete definition, please consult the Odc.wsdl file):

Java Lib Description

com.sap.odc.mer.api.pos.SearchMERSubscriptionParameters	
Input parameters	
ServiceProviderCode (String)	The code of the merchant
OfferCode (String)	The code of the offer
Output/Result parameters	
Subscription (TLVSubscription)	The details of the subscription that will contain the data for the following fields which contain paramters (List<TLVParameter>): - SharedParameters (List<TLVParameter> - ChargeActivations (List<TLVChargeActivation>) o Parameters (List<TLVParameter>

TLV C-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_PARAMETER_OP**	**0xBA**	var	var	C
1	**TAG_ALEA**	**0x09**	**0x0008**	var	P
2	**TAG_SERVICE_PROVIDER_ID**	**0x1A**	**0x0002**	var	P
3	**TAG_OFFER_ID**	**0x3E**	**0x0002**	var	P

TLV R-APDU Description

Idx	TAG NAME	TAG	LENGTH	VALUE	C/P
0	**INS_SEARCH_MER_SUBSCRIPTION_PARAMETER_RESULT**	**0xBB**	var	var	C
1	**TAG_SUBSCRIPTION (*)**	**0x51**	**var**	var	C

(*) defined in previous section

Important Note

Considering the complexity of the Subscription, we recommend the user to use the high-level java API for getting information from the embedded subscriptions.

The involved API are:

- **SearchMERSubscriptionInfo()**
- **SearchMERSubscriptionCounters()**

- **SearchMERSubscriptionParameters()**