Oracle® Communications Network Charging and Control

Charging Control Services Technical Guide

Release 15.0.0

October 2023

Copyright

Copyright © 2023, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government’s use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

About This Document vii
Document Conventions viii

Chapter 1

System Overview 1

Overview 1
Introduction to Charging Control Services 1
How CCS Fits Into the Network 3
CCS Components Overview 4
Voucher and Wallet Server and CCS 10
CCS on a Clustered platform 14
Configuring Services 15
Subscriber Accounts and Wallet Management 16
Security 23
About Secure SSL Connection to the Database 25
Calling Card Services 26
Rating and Charging 28
Periodic Charges 31
Recharges 37
Promotions 38
Notifications 39
EDRs 43

Chapter 2

Configuration 47

Overview 47
Configuration Overview 47
Configuring the Environment 49
eserv.config Configuration 50
Configuring acs.conf for the SLC 53
Setting up the Screens 60
Defining the Screen Language 64
Defining the Help Screen Language 65
Configuration Through the ACS Screens 66
User Interface-Based Configuration Tasks 67
Configuring VWS processes for CCS 68
Configuring CCS Macro Nodes 69
Switch Configuration for the UATB Node 71
Voucher Status Report Configuration 73
CCP Configuration 74

Chapter 3

Background Processes on the SMS 87

Overview 87
CHECK_PC_DELETION 88
acsCompilerDaemon 88
ccsBeOrb 89
ccsCB10HRNAES 108
ccsCB10HRNSHA 108

ccsCDRFileGenerator 108
ccsCDRLoader 111
ccsCDRTrimDB 129
ccsCDRTrimFiles....130
ccsChangeDaemon....131
ccsExpiryMessageLoader 137
ccsExternalProcedureDaemon....141
ccsLegacyPIN....142
ccsPeriodicCCRecharge 142
ccsPeriodicCharge 144
ccsProfileDaemon 152
ccsReports....163
ccsWalletExpiry 167
libccsCommon....170
VoucherRedeemFail Files 170

Chapter 4

Background Processes on the SLC 173

Overview....173
BeClient....173
PIClientIF....179
ccsActions 180
ccsCB10HRNAES 184
ccsCB10HRNSHA....184
ccsMacroNodes....184
ccsSvcLibrary 196
libccsClientPlugins 205
libccsCommon....207

Chapter 5

Background Processes on the VWS 209

Overview....209
beVWARS 210
ccsActivationCharge....212
ccsBadPinPlugin....213
ccsCB10HRNAES 214
ccsCB10HRNSHA....215
ccsBeAvd....215
ccsExpiryMessageGenerator 215
ccsLegacyPIN....216
ccsMFileCompiler 216
ccsNotification 220
ccsSLEEChangeDaemon....223
ccsPDSMSPlugin 232
ccsRewardsPlugin....234
ccsPMXPlugin 238
ccsVWARSActivation 241
ccsVWARSAmountHandler....243
ccsVWARSExpiry....244
ccsVWARSNamedEventHandler 252
ccsVWARSPeriodicCharge 257
ccsVWARSQuota 262
ccsVWARSRechargeHandler....263
ccsVWARSReservationHandler 263

ccsVWARSVoucherHandler 270
ccsVWARSWalletHandler 274
ccsVWARSPolicyHandler 276
ccsWLCPlugin 276
ccsPolicyPlugin 277
cmnPushFiles 277
libccsCommon 282
libccsVWARSUtils 292

Chapter 6

Tools and Utilities 297

Overview 297
ccsAccount 297
ccsBeResync 322
ccsBatchCharge 341
ccsDomainMigration 344
ccsMFileDump 349
ccsProfileBulkUpdate 352
ccsVoucherStartup.sh 353
CCS Balance Top Up Suite 353
CCS Balance Top Up MSISDN Files 355
CCS Balance Topup Rule Scripts 357
dwsublist.sh 361
Example Balance Top Up Rule Execution 363

Chapter 7

Real-Time Notifications 367

Overview 367
Real-Time Notifications 367
Notification Construction 369

Chapter 8

About Installation and Removal 373

Overview 373
Installation and Removal Overview 373
Checking the Installation 373

About This Document

Scope

The scope of this document includes all the information required to install, configure and administer the Charging Control Services (CCS) application.

Audience

This guide was written primarily for system administrators and persons configuring and administering the CCS application and the VWS. However, sections of the document may be useful to anyone requiring an introduction to the application.

Prerequisites

A solid understanding of Unix and a familiarity with IN concepts are an essential prerequisite for safely using the information contained in this technical guide. Attempting to configure or otherwise alter the described system without the appropriate background skills, could cause damage to the system; including temporary or permanent incorrect operation, loss of service, and may render your system beyond recovery.

This manual describes system tasks that should only be carried out by suitably trained operators.

Related Documents

The following documents are related to this document:

- *Charging Control Services Alarms Guide*
- *Voucher and Wallet Server Technical Guide*
- *Charging Control Services User's Guide*
- *Customer Care Portal User's Guide*
- *Subscriber Profile Manager User's Guide*
- *Voucher Manager User's Guide*
- *Promotion Manager User's Guide*
- *Advanced Control Services User's Guide*
- *Advanced Control Services Technical Guide*
- *Feature Nodes Reference Guide*
- *Service Management System Technical Guide*
- *Service Management System User's Guide*
- *Service Logic Execution Environment Technical Guide*

Document Conventions

Typographical Conventions

The following terms and typographical conventions are used in the Oracle Communications Network Charging and Control (NCC) documentation.

Formatting Convention	Type of Information
Special Bold	Items you must select, such as names of tabs. Names of database tables and fields.
Italics	Name of a document, chapter, topic or other publication. Emphasis within text.
Button	The name of a button to click or a key to press. **Example:** To close the window, either click **Close**, or press **Esc**.
Key+Key	Key combinations for which the user must press and hold down one key and then press another. Example: **Ctrl+P** or **Alt+F4**.
`Monospace`	Examples of code or standard output.
`Monospace Bold`	Text that you must enter.
variable	Used to indicate variables or text that should be replaced with an actual value.
menu option > menu option >	Used to indicate the cascading menu option to be selected. Example: **Operator Functions > Report Functions**
hypertext link	Used to indicate a hypertext link.

Chapter 1

System Overview

Overview

Introduction

This chapter provides a high-level overview of the application. It explains the basic functionality of the system and lists the main components.

It is not intended to advise on any specific Oracle Communications Network Charging and Control (NCC) network or service implications of the product.

In this Chapter

This chapter contains the following topics.

Introduction to Charging Control Services 1
How CCS Fits Into the Network 3
CCS Components Overview 4
Voucher and Wallet Server and CCS 10
CCS on a Clustered platform 14
Configuring Services 15
Subscriber Accounts and Wallet Management 16
Security 23
About Secure SSL Connection to the Database 25
Calling Card Services 26
Rating and Charging 28
Periodic Charges 31
Recharges 37
Promotions 38
Notifications 39
EDRs 43

Introduction to Charging Control Services

Description

The Charging Control Services (CCS) is a prepaid and post-paid service, which allows customers greater flexibility and control over their billing methods and telephony services in general. It provides options for customers with low credit ratings, at the same time as furnishing all customers with a high-quality and adaptable range of services. This allows the service provider to customize call-processing functionality according to factors such as:

- Geography
- Demographics
- Resources
- User preference

How It Works

CCS is installed and run as a network service by a Telecommunications Provider (telco). This service allows the telco to create:

- Subscriber accounts and wallets
- Product types to be associated with the subscriber wallet

Each product type may be linked to a rate table, each of which may have many tariff options. CCS uses a three-tier tariff scheme.

1 Basic tariffs use the flexible geography sets to determine calling areas.
2 Weekly tariffs are available to override the basic tariffs where applicable.
3 Holiday tariffs may be set to override both basic and weekly tariffs.

Subscriber Access

CCS supports several possible access points for subscriber, including:

- Fixed line
- Mobile line
- IP connection
- Carrier pre-select charging
- Home Zone / Office Zone

Business Process Logic

CCS provides the facility to define Business Process Logic (BPL) tasks. Each BPL task defines a set of actions that, when executed, perform a specific business process for a subscriber, for which the subscriber may optionally be charged.

BPL tasks are defined by the service provider. Each BPL task has an associated control plan that can be started through one of the following:

- CCS screens
- Provisioning Interface (PI)

For more information about BPL task definition, see the *Task Management* chapter in *Charging Control Services User's Guide*.

Periodic Charges

Periodic charges enable the telco to set regular subscriber charges. For example, you can define periodic charges for providing a phone service, or for rental of services and equipment. Periodic charges can also be configured for sending notifications and for performing voucher type recharges.

Periodic charges are associated with product types, and must be subscribed to by subscribers.

Notes:

- Each time a periodic charge occurs, it is logged in an EDR.
- This functionality is available only if you purchase the Periodic Charges license. For more information about the screens configuration, see *Charging Control Services User's Guide*.

Vouchers

CCS provides voucher functionality. This functionality is described in *Voucher Manager Technical Guide*.

How CCS Fits Into the Network

Introduction

There are four major functional layers in the Oracle Communications Network Charging and Control (NCC):

1. Service Management
2. Service Applications
3. Context Management
4. IN Control

Service Management

Centralized management and an extensive set of service reporting and alarm management functionality is provided to ease the administration of the entire platform.

Service Applications

This layer provides a graphical control plan management and provisioning interface for users. A rich set of service features and powerful call routing functionality is available.

Context Management

This layer manages each (message) call event coming into and going from the service application layer. Every message represents an event happening during a call; the message must be received from the underlying network and passed to the service application, and vice versa.

This layer is designed to maintain integrity, simplify management, and ensure high performance when managing multiple messages from multiple underlying networks to multiple applications.

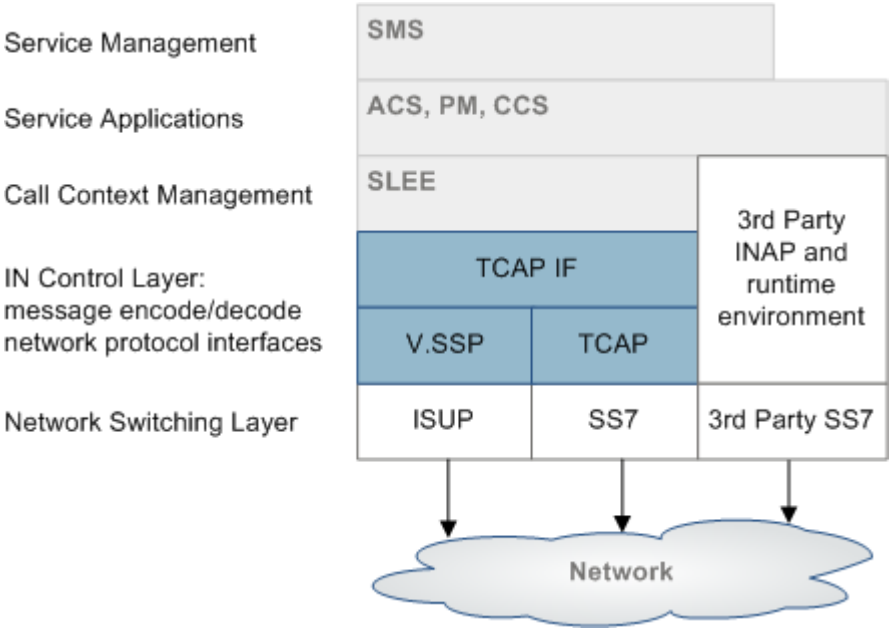
IN Control

This layer enables the service application layer to be available on networks with multiple different communications protocols (for example, INAP, ISUP, H.323). NCC provides generic interfaces for H.323, ISUP and INAP.

Depending on the underlying network protocol, these interfaces translate call events and messages from the network into INAP messages that can then be sent through the context management layer to the service application layer. The reverse happens for messages coming the other way.

Diagram

Here is an example showing how CCS fits into the application layer.



CCS Components Overview

Platform components

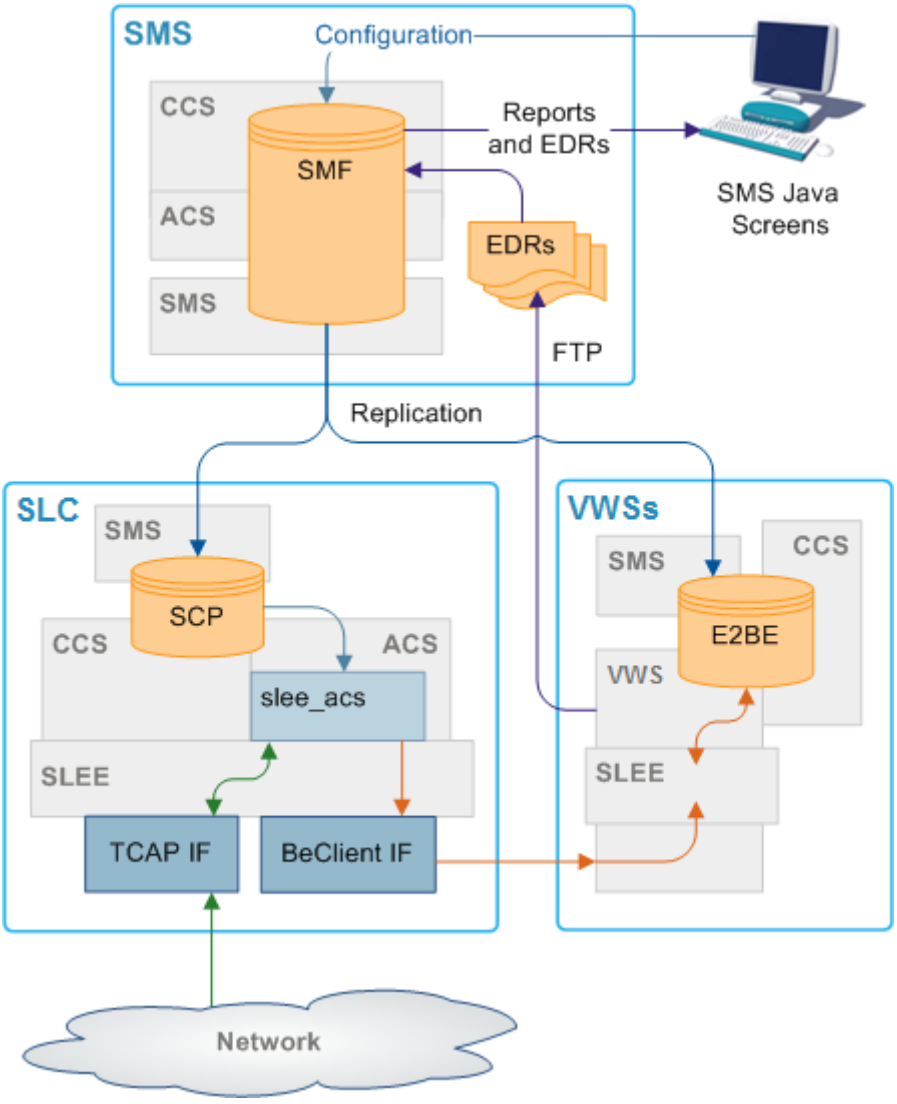
This table describes the main components in CCS.

Note: CCS is installed on all three machines.

Component	Role
SMS	The central management system of the platform. It hosts the authoritative configuration and subscriber database (SMF), and provides access to the external world using provisioning interfaces and using a graphical user interface (SMS screens). It is responsible for keeping all platforms in sync, and acts as a central collection point for alarms and statistics of the entire platform.
SLC	Performs all actual call switching. It interfaces with the telephony network and executes the service logic for each service. It also interfaces with the Voucher and Wallet Servers to ensure that calls are charged in real-time.
VWS	The Voucher and Wallet Server hosts the subscriber balances and acts as the rating engine. It processes incoming rating and charging requests and maintains wallet data. For more information about Voucher and Wallet Servers, see *Voucher and Wallet Server and CCS* (on page 10).

System diagram

Here is an example of how CCS fits into a standard install of NCC software.



Supporting applications

Some of the components of CCS are supplied by the other applications.

Application	Role	Further information
SMS	Provides the base system management functionality including the SMS Java administration screens and centralized data storage and replication, including: • EDRs • Alarms • Statistics	*Service Management System User's Guide* *Service Management System Technical Guide*

Application	Role	Further information
ACS	Provides call and SMS processing and control, customer/service provider management and control plan creation. ACS functionality is extended by CCS plug-ins (macro nodes, configuration and libraries).	*Advanced Control Services User's Guide* *Advanced Control Services Technical Guide*
VWS	Provides billing facilities. May be replaced by a third-party billing engine. VWS database is the VWS database, it also holds CCS data.	*Voucher and Wallet Server Technical Guide*

Subsystems used by CCS

The main subsystems used by CCS are:

- Replication (provided by SMS)
- ACS and CPE (for call processing)
- EDR generation and file transfer
- SMS Java administration screens and optional PI commands
- VWS (for charging, and subscriber account and wallet management)

Note: Each subsystem (except the SMS administration screens) must be configured to support CCS. The SMS administration screens are automatically configured when CCS is installed.

CCS and ACS

Some aspects of the Advanced Control Services (ACS) service are available to the CCS operator, providing call-processing functionality to the CCS base service.

The core ACS functionality may be used by operators or service providers in conjunction with the CCS service. This provides additional value and adds processing capability. For example, personal or global barring lists, special PIN accessed functionality, or speed dial codes.

ACS requires some configuration to enable CCS to operate correctly.

For more information about:

- Configuring ACS for CCS, see *Configuration* (on page 47)
- ACS, see *Advanced Control Services Technical Guide*

CCS Control Plans

Calls using the CCS service are routed to a terminating point using a control plan. A control plan is a service-logic flowchart that consists of a collection of feature nodes that are used to define the call flow. Each feature node defines a particular decision point or action that determines where next to route a call.

Note: Credit transfers require a special control plan called CREDIT_TRANSFER. This control plan is installed by default, and is required to process credit transfer commit requests. For more information about credit transfers, see the *Transfer Management* chapter in *Charging Control Services User's Guide*.

For more information about CCS feature nodes, see *Feature Nodes Reference Guide*.

You can also create global CCS control plans. Global control plans enable the operator to screen calls before the customer's control plans are applied. Global control plans are owned only by the operator and are automatically assigned to the default operator customer.

Global control plans are associated with a specific service. If you create a global control plan and associate it with the CCS service, the control plans' service logic is applied to calls for all customers who use the CCS service.

For more information about managing control plans, see *Control Plan Editor User's Guide*.

CCS and VWS

The CCS base service uses a fault-tolerant Voucher and Wallet Server, known as VWS. Keeping the Voucher and Wallet Server logically separate from the call-processing engine allows it to be used by multiple clients.

CCS provides call control and business rules. It handles:

- Subscribers
- Tariffing
- Vouchers
- Money
- Provisioning
- Credit cards
- Relationship between subscriber accounts and wallets

CCS uses the VWS for executing financial functions for CCS and managing wallets and balances. Familiarity with the VWS design and structure is assumed. For more information about the VWS, see *Voucher and Wallet Server Technical Guide*.

Note: A third party domain may be used instead of the VWS to service billing requirements.

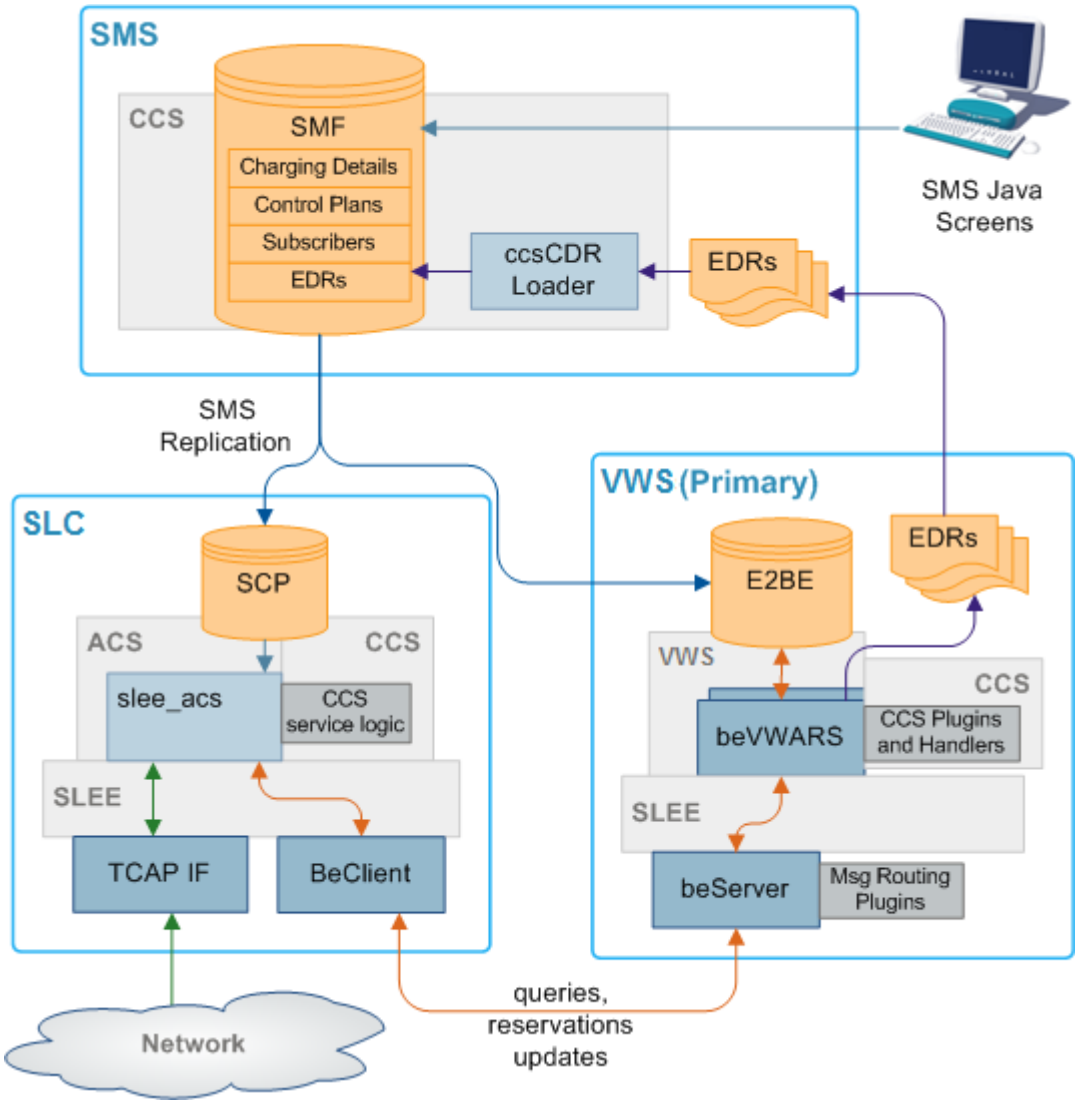
CCS components

CCS has these types of components:

- Data (subscribers, charges, vouchers, promotions)
- CCS Java administration screens (enables users to manage data)
- CCS plug-ins to Voucher and Wallet Server (execute tariffing and business rules)
- CCS plug-ins to ACS for call control (includes CCS feature nodes for charging control plans)
- Command-line tools and utilities

Component diagram

Here is an example showing the main components of CCS.



Component description

This table describes the main components in CCS.

Component	Role	Further information
SMS Java Administration screens	These administration screens provide a GUI for configuring CCS.	*NCC Charging Control Services User's Guide*
SMF database	The main database on the SMS. This database holds data for CCS and the other applications installed alongside it.	*NCC Service Management System Technical Guide*
SCP database	The databases on the SLCs. They hold a subset of the data in the SMF database.	
E2BE database	The databases on the Voucher and Wallet Servers. They hold a subset of the data on the SMF. They primarily hold VWS and CCS data.	*NCC Voucher and Wallet Server Technical Guide*

Component	Role	Further information
ccsCDRLoader	Inserts EDRs into the SMF so the SMS screens can be used to view call and system activity.	*ccsCDRLoader* (on page 111)
slee_acs	The slee_acs process handles call processing on the SLC. Compiled control plans provide the call process configuration.	*NCC Advanced Control Services Technical Guide*
CCS Service Logic	slee_acs is extended by CCS-specific functionality which enables charging control plans.	*NCC Control Plan Editor User's Guide*
SLEE	The Service Logic Execution Environment routes calls to the slee_acs and to other machines through the SLEE interfaces (TCAP IF and BeClient IF).	*NCC Service Logic Execution Environment Technical Guide*
TCAP IF	The TCAP IF is the interface between the SLEE and the TCAP stack.	*NCC XML TCAP Interface Technical Guide*
BeClient IF	The BeClient interface processes requests from the call processor to the Voucher and Wallet Servers.	*Voucher and Wallet Server Technical Guide*
beServer	The beServer handles all incoming requests to the Voucher and Wallet Servers.	*NCC Voucher and Wallet Server Technical Guide*
beVWARS	The beVWARS handles all actions involving vouchers, wallets and accounts. beVWARS is extended using CCS plug-ins.	*NCC Voucher and Wallet Server Technical Guide*

CCS service logic

The CCS service logic is provided to extend the ACS slee_acs process to provide charging and billing functions. This table describes the plug-in libraries which provide the CCS service logic.

Plug-in Library	Purpose
ccsSvcLibrary (on page 196)	The CCS service library handles the initial call setup for calls which will use CCS functionality. It determines which control plan to use, and populates any necessary profile data.
ccsMacroNodes (on page 184)	The CCS macro nodes library provides the CCS macro nodes which are used in control plans which use CCS.
ccsActions (on page 180)	The CCS chassis action library provides functions which are used when ccsSvcLibrary requires an action outside slee_acs. This library is primarily used for billing actions which are completed by the VWS.

For more information about how these libraries are included in slee_acs, see *Configuring acs.conf for the SLC* (on page 53).

Note: If a third-party VWS is used, a different chassis action library will be provided. For more information about these chassis action libraries, see the technical guide for the application which provides connectivity to the third-party Voucher and Wallet Server.

Replication

Replication is the main method used to transfer relevant data from the main SMF database on the SMS to the databases which are used for specific functions. Each replication point (node) must be configured in SMS before it can be used in CCS.

For more information about replication, see *Service Management System Technical Guide*.

CCS replication

For CCS, replication forwards data from the SMF to the SCP and E2BE databases.

The data replicated to the SCP are:

- Subscriber data
- ACS compiled control plans

The data replicated to the E2BE are:

- Tariffs and tariff rate tables
- CCS Mfile data
- Subscriber and wallet data

Note: Some of the CCS plug-ins for VWS require additional data from the SMF database on the SMS. These tables and their replication configuration are installed with the ccsSms package.

CCS-VWS Protocol overview

The new CCS-VWS protocol is built upon an extensible self-describing message format called Escher. The new protocol is easily extensible, versioned, and allows additions without breaking backward compatibility. The CCS-VWS protocol definition is defined for internal use only.

Voucher and Wallet Server and CCS

Domains

CCS provides the facility to control which service is provided by which network element using domains.

A domain defines what functionality CCS uses a set of one or more domain nodes for. Domain nodes are network elements which provide one or more of the following functions:

- Rating
- Billing
- Wallet management
- Voucher management

An example of a domain would be a pair of NCC Voucher and Wallet Servers.

Domains enable CCS to separate traffic for a dedicated service such as voucher redemption.

For more information about configuring domains, see *Charging Control Services User's Guide*.

Distributed Wallet Management

You can distribute wallet management across two domains. The wallet management functionality is split between the following two elements:

- Charging management
- Tracking management

A domain can be configured to support one or both of these elements. This allows chargeable balances to be held on the charging domain, and fraud and expense balances to be held separately on a tracking domain.

Note: Tracking domains can only be configured for a VWS domain type. Charging domains can be configured for any domain type.

Domain Types

Domain types enable CCS to handle groups of domain nodes that share a common technology. This can reflect the communication protocol, and/or make and model of the node.

Examples: The following are domain types:

- VWS
- DIAMETER
- Intec

For more information about configuring these domain types, see Domain.

Default domain type

The default domain type for a call is set by the service loader library which loads the control plan for the call. For example: ccsSvcLibrary sets the default domain to 1.

Overriding default domain types

The default domain type for ccsSvcLibrary can be overridden using one of the following:

- The **eserv.config** parameters are one of the following:
  - SubscriberDomainType
  - VoucherDomainType
- The **Domain** drop down list on the **Wallet** option on the Edit Subscriber screen.

Notes:

- These overrides only work for the ccsSvcLibrary. If the call is being processed using a different service loader library, see the application's technical guide for details of how the domain type is set.
- If the call is being processed by ccsSvcLibrary using a service loader plug-in, see the plug-in application's technical guide for details of any default domain type setting and overriding.

Changing domains during call processing

The Set Active Domain feature node enables the domain type to be changed at any point within a control plan.

For example, if TUS is installed with the default Voucher Domain type as '2' (for TUS), then the domain can be changed mid call to VWS and vice versa using the Set Active Domain feature node.

For more information about the Set Active Domain feature node, see *Feature Nodes Reference Guide*.

CCS and VWS

The CCS base service uses a fault-tolerant Voucher and Wallet Server, known as VWS. Keeping the Voucher and Wallet Server logically separate from the call-processing engine allows it to be used by multiple clients.

CCS provides call control and business rules. It handles:

- Subscribers
- Tariffing
- Vouchers
- Money
- Provisioning
- Credit cards
- Relationship between subscriber accounts and wallets

CCS uses the VWS for executing financial functions for CCS and managing wallets and balances. Familiarity with the VWS design and structure is assumed. For more information about the VWS, see *Voucher and Wallet Server Technical Guide*.

Note: A third party domain may be used instead of the VWS to service billing requirements.

Subscribers and wallet management

CCS provides a number of services with VWS. They include:

- Balance check
- Subscriber management and wallet charging
- Business process logic
- Merge wallets facility
- Wallet grace periods
- Voucher and credit card recharges
- Automatic deletion of redeemed vouchers
- Wallet and balance expiry and subscriber notification
- Product type updates and notifications
- EDR generation

Diagram

Here is an example of how the VWS handles requests from CCS on an SLC to a VWS.

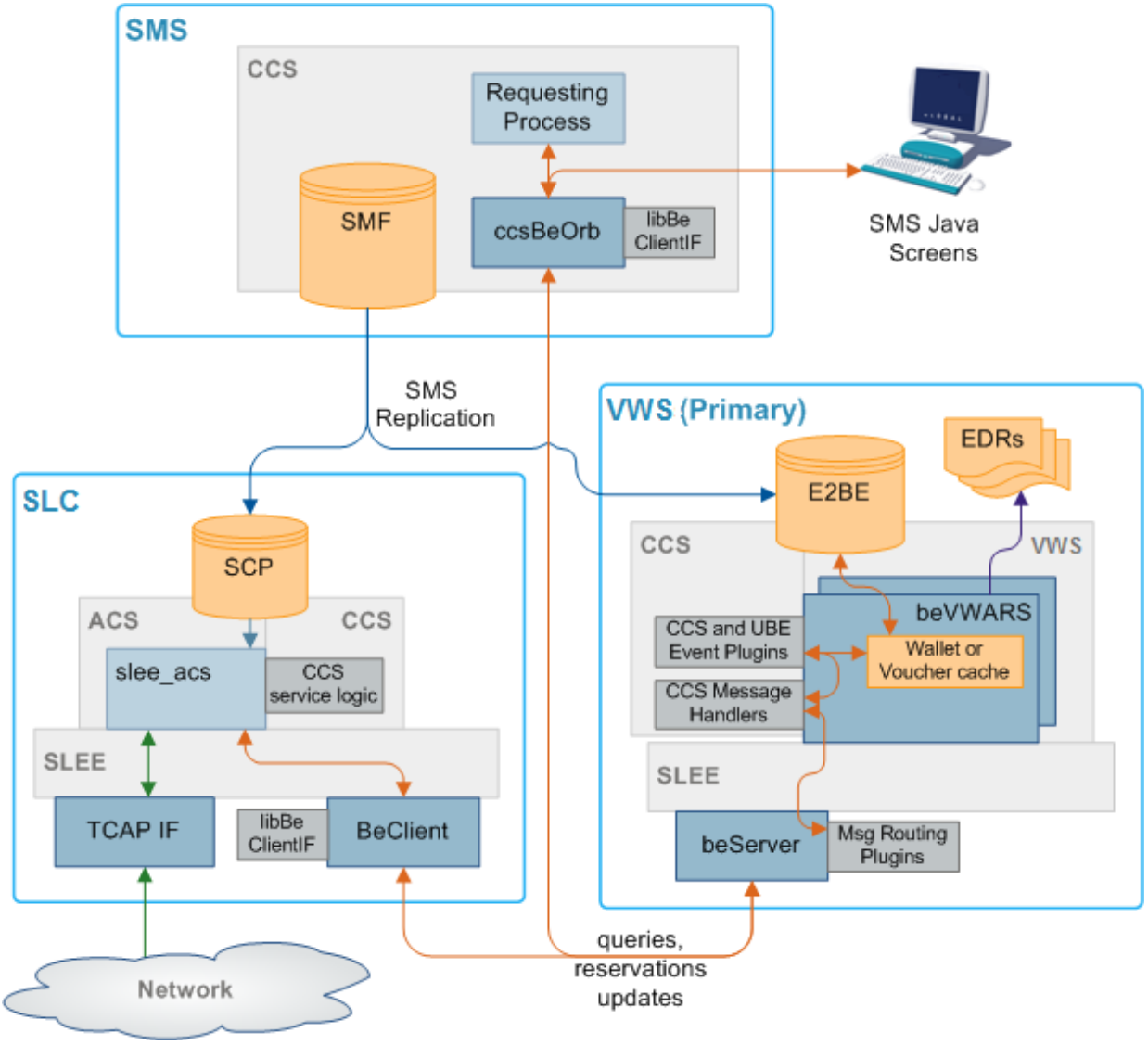
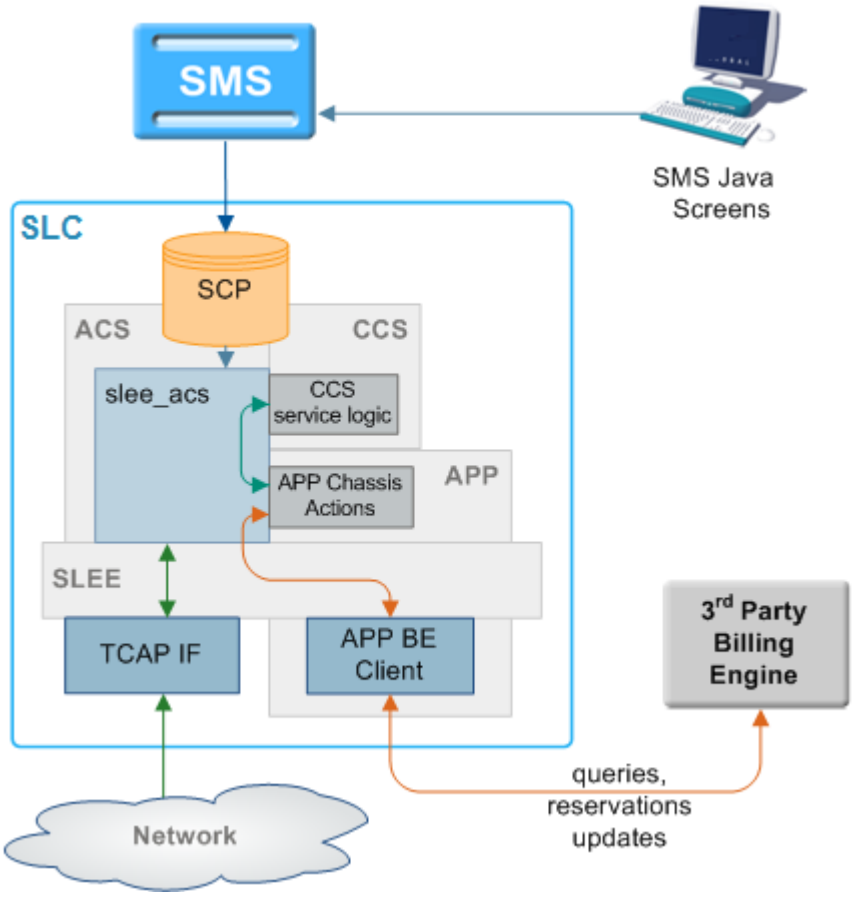


Diagram - Third party Voucher and Wallet Servers (VWS)

This diagram shows the CCS components involved in interaction with third party Voucher and Wallet Servers.

Note: For each type of third party VWS, a different extension will be installed to work with CCS.



Starting and stopping the VWS

The VWS runs on top of the SLEE, so the normal SLEE start/stop commands should be used on the VWS machine using the ebe_oper user, to start and stop it.

The VWS will go through several phases before making itself available for calls, the duration of these phases depends on the speed of the network link to the other Voucher and Wallet Server in the pair and the length of time the Voucher and Wallet Server has been down. The VWS will not enable itself until it is closely synchronized with the other Voucher and Wallet Server (which will be acting as primary) so as to minimize the problems caused by timing delays in the synchronization process when a swap from secondary to primary occurs. If the partner Voucher and Wallet Server cannot be contacted then the recovering Voucher and Wallet Server will enable after a configurable number of connection attempts.

For more detail about the VWS design, implementation and operation see *Voucher and Wallet Server Technical Guide*.

CCS on a Clustered platform

Introduction

CCS can be integrated with SMS 3.0 which introduces support for a clustered SMS configuration. Using a clustered configuration means that critical management processes can be executed on multiple machines minimizing the amount of downtime of the overall system.

CCS/VWS management processes are split into three categories of availability:

1. Single node services with automated failover
2. Multi-node services
3. Single node services with manual restart

Single Node Services with Automated Failover

The EDR management process is only executed on a single node, even when the SMS is in a clustered configuration. The process fails over to an alternate node within 20 seconds.

Multi-Node Services

The following CCS/VWS processes operate concurrently on all nodes in a cluster:

Process	Description
CLI-DN Daemon	This allows calling and called numbers to be cross-referenced in order to begin determining the rate for a call.
ccsBeOrb	This is the CCS CORBA gateway to the Voucher and Wallet Server.
ccsCDRLoader	Loads EDR files into the SMF database.
ccsRewardsBatch	Processes rewards requests from the VWS.

Single Node, Manual re-start services

The following processes require a manual restart in case the node executing the process fails.

- ccsAccount
- ccsVoucher
- ccsBeResync

Configuring Services

Introduction

CCS can support more than one service at the same time. Consequently, each service must be defined so CCS can determine which service to apply to each call.

Configuration overview

Configuring services involves:

- SLEE and slee_acs routing
- Defining capabilities
- Defining tariffs
- Defining product types
- Creating appropriate control plans

SLEE and slee_acs routing

Calls are routed to slee_acs over the SLEE. Each call has:

- A service key
- An originating number (CLI or MSISDN)

- A terminating number (DN or MSISDN).

The service triggers to different service loaders within slee_acs depending on:

- Service key
- Terminating number

The relationship is defined in **acs.conf**.

Capabilities

Capabilities enable calls sent to the same service key to be handled differently depending on the bearer capability in their IDP. For example, Voice and Video for same service key can have different control plans and tariff plans.

CCS screens configure IDP to capability routing. You can set up a global capability which applies to all product types or a capability can have a specific control plan (and tariff plan if specified).

Services are defined in **acs.conf** using the ServiceEntry configuration. The first argument in the ServiceEntry matches to Service field in a capability. Default control plan is invoked if a subscriber cannot be loaded.

Example:

```
ServiceEntry (CCS,ccsSvcLibrary.so)
```

For more information about ServiceEntry configuration, see *Advanced Control Services Technical Guide*.

Note: Default control plan is used if no subscriber can be loaded (and therefore CCS cannot locate a control plan by product type).

Bearer capabilities

Bearer capability specifies a requested service: packet or circuit mode, data rate, type of information content. The bearer capability is made up of a number of different bits, but the number you enter in the capability screen is actually the InitialDP itc field (information transfer capability).

This table shows some capabilities and their general uses.

Capability	Description
0	Speech
8	Unrestricted Digital Information
9	Restricted Digital Information
16	3.1 Khz Audio
17	Unrestricted Digital Information with Tones/Announcements
24	Video

Note: These capabilities are shown in decimal.

Subscriber Accounts and Wallet Management

Introduction

Actions regarding subscriber accounts and wallets can be completed by either CCS processes or Voucher and Wallet Server processes. The CCS processes complete actions in the following areas:

- Sending wallet and voucher requests to the Voucher and Wallet Server
- Updating subscriber account and wallet expiry and activation details in the SMF

- Updating subscriber's account and product type details
- Generating short messages which are sent to subscribers reminding them that their wallet or balance will shortly run out, or informing them of any balance or product type changes

For more overview information about subscriber accounts and wallets, see *Charging Control Services User's Guide*.

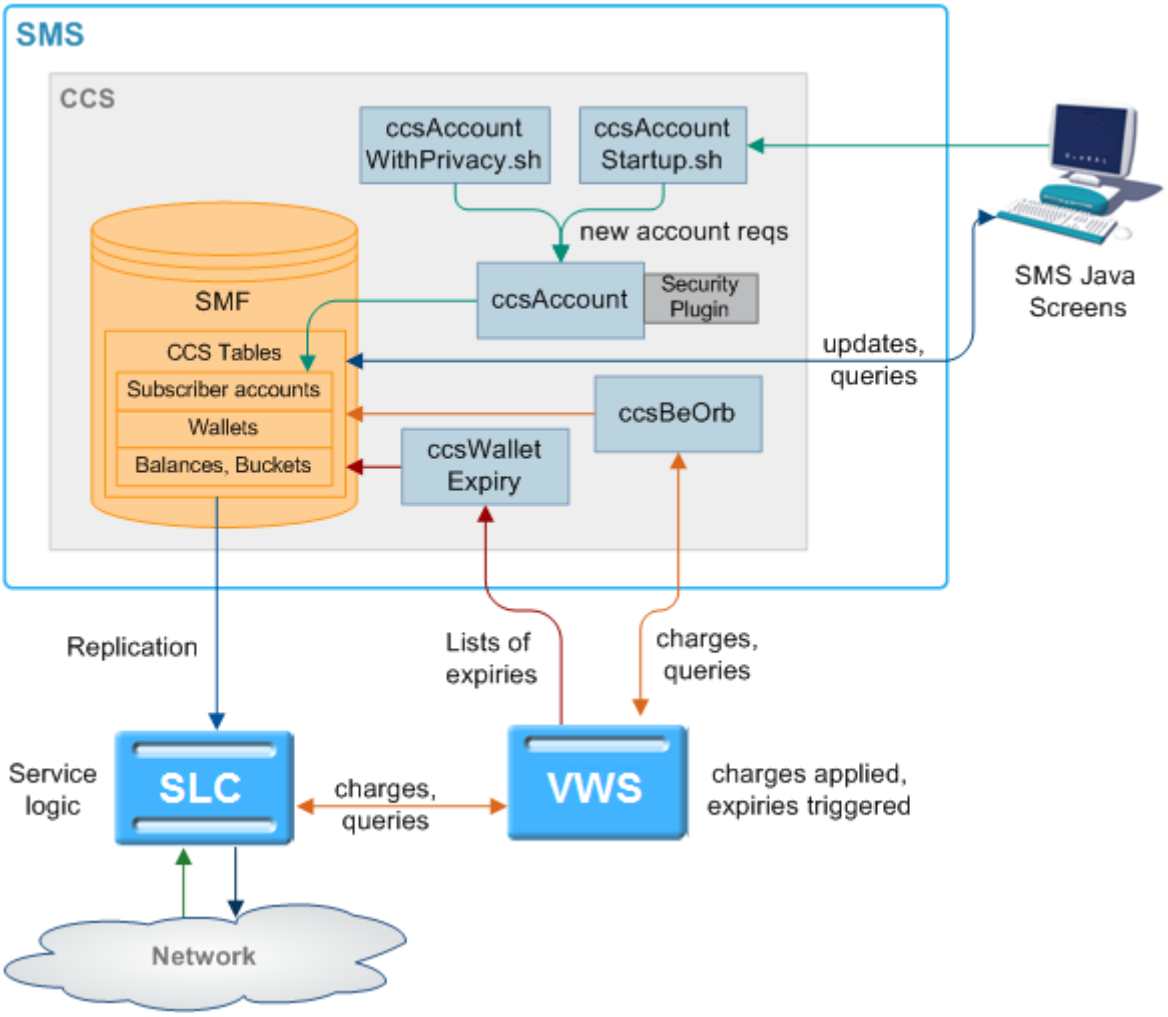
CCS plug-ins for the VWS

If the platform uses a Voucher and Wallet Server, the VWS processes handle the VWS-end of wallet or voucher related actions. CCS functionality is provided by adding plug-in libraries to the VWS processes on the VWS. The message and wallet handler plug-ins on the VWS are installed by the ccsBe package. These are explained in detail in *Background Processes on the VWS* (on page 209).

For more information about the VWS processes involved in subscriber account and wallet management, see *Voucher and Wallet Server Technical Guide*.

Diagram

This diagram shows some elements relating to subscriber account, wallet and bucket creation and expiry/removal.



For more information about:

- Charging, see *Rating and Charging* (on page 28)
- Expiry, see *Voucher and Wallet Server Technical Guide*

Subscriber accounts and wallet processes

This table describes the main processes involved in subscriber and wallet management.

Process	Role	Further information
ccsAccount	Generates batches of subscriber accounts.	*ccsAccount* (on page 297)
ccsAccountStartup.sh	Startup script for ccsAccount.	*Startup - ccsAccountStartup.sh* (on page 298)
ccsAccountWithPrivacy.sh	Startup script for ccsAccount with encryption.	*Startup - ccsAccountWithPrivacy.sh* (on page 298)
Security modules	Used by ccsAccount when started by ccsAccountWithPrivacy.sh.	*Authenticating modules* (on page 23)
ccsBeOrb	Handles communication between SMS screens and VWSs.	*ccsBeOrb* (on page 89)
libBeClientIF	This library provides common functions for the connection with the VWS VWSs.	*Voucher and Wallet Server Technical Guide*
ccsExpiryMessage Generator	ccsExpiryMessageGenerator generates a list of wallets or balances which will expire shortly and writes it to a file on the VWS VWS.	*ccsExpiryMessageGenerator* (on page 215)
cmnPushFiles	cmnPushFiles forwards the expiry list file to the SMS.	*cmnPushFiles* (on page 277)
cmnReceiveFiles	cmnReceiveFiles accepts the expiry list file from cmnPushFiles and writes it to the directory indicated by cmnPushFiles.	*Service Management System Technical Guide*
ccsExpiryMessage Loader	ccsExpiryMessageLoader sends short messages to subscribers to warn them that their wallet or balance will expire shortly.	*ccsExpiryMessageLoader* (on page 137)
ccsWalletExpiry	ccsWalletExpiry processes CCS updates to the subscriber and wallet expiry tables on the SMF.	*ccsWalletExpiry* (on page 167)

Wallets and VWS VWSs

If CCS is using Voucher and Wallet Servers (VWSs), each wallet is created on a specific VWS. To perform an action on a wallet or its balances and buckets, the requesting process must know which VWS to send the message to. This information is stored in a reference table which is stored on the SMS and replicated to the SLC.

Generating Accounts

This table describes the process ccsAccount follows to create CCS subscribers and wallets by batch.

Stage	Description
1	On the SMS, ccsAccount logs into the SMF database using Oracle user ID ccs_admin and creates rows in the following tables: • CCS_ACCT

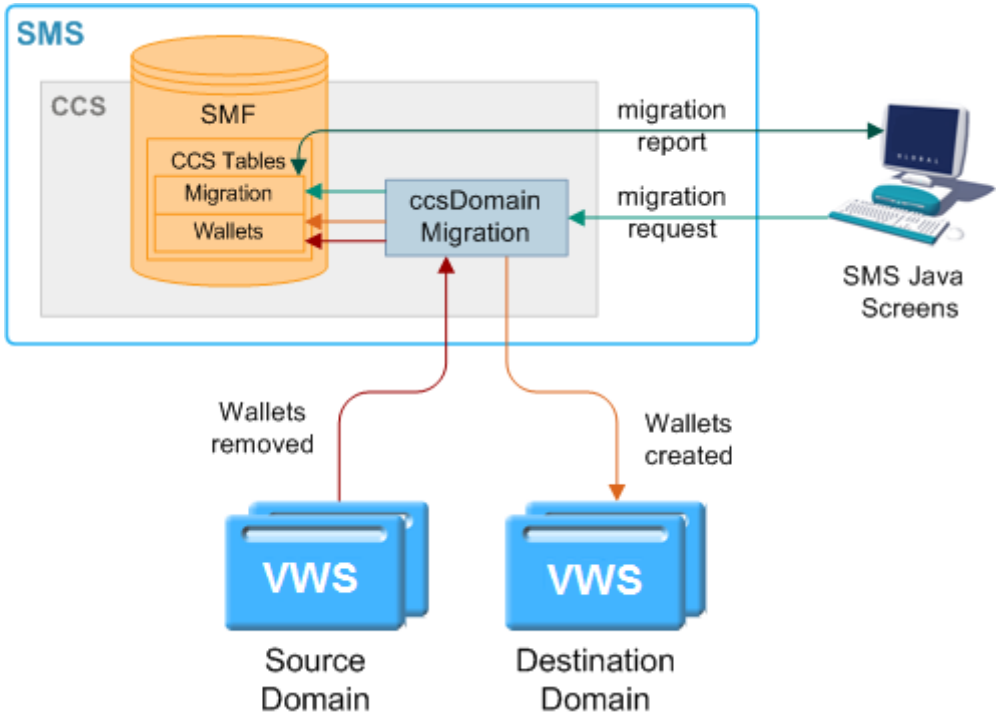
Stage	Description
	• CCS_ACCT_REFERENCE • CCS_ACCT_ACCT_REFERENCES • CCS_ACCT_HIST_INFO The rows are entered by calling the methods of packages on the SMS.
2	ccsAccount then requests that the Voucher and Wallet Server make the Wallets for the Subscribers by making rows in: • BE_WALLET • BE_BALANCE • BE_BUCKET
3	The CCS_* rows are replicated out to the VWSs and SLCs by replication.

Notes:

- ccsAccount may also create accounts using the privacy setting. For more information about this process, see *Generating account numbers* (on page 26).
- ccsAccount must be able to contact the Voucher and Wallet Servers at all times. If the connection drops to one of the pair it will switch over to the secondary. If the secondary also goes down, ccsAccount will try to re-send its request a configurable number of times before giving up.
- All the wallets are created on one VWS only. If the VWS pair ID is not specified, it will pick the VWS with the lowest ratio of 'Maximum Accounts' (java screens, Subscriber Management->Domain) to the actual number of wallets on a VWS.

Wallet migration diagram

This diagram shows the elements involved in migrating wallets from one Voucher and Wallet Server to another.



Wallet migration process descriptions

This table describes the main processes involved in migrating wallets from one Voucher and Wallet Server pair to another.

Process	Role	Further information
ccsDomainMigration	ccsDomainMigration manages the migration of wallets from one VWS to another. It connects to beServer on the Voucher and Wallet Servers.	*ccsDomainMigration* (on page 344)
libBeClientIF	This library provides common functions for the connection with the VWSs.	*Voucher and Wallet Server Technical Guide*

Wallet migration process

This table describes how wallets are migrated from one Voucher and Wallet Server pair to another using wallet migration.

Stage	Description
1	The user configures a migration using the **UBE Account Balancing** tab on the Subscriber Management screen and clicks **Confirm** on the Confirmation Dialog prompt. For more information about the **UBE Account Balancing** tab, see *Charging Control Services User's Guide*.
2	The screens trigger the ccsDomainMigration daemon using its startup script: ccsDomainMigrationStartup.sh
3	ccsDomainMigration reads configuration from **eserv.config**.
4	ccsDomainMigration checks for a lockfile (the lockfile is specified by the *lockFile* (on page 348) parameter or the default is used). If the lockfile is present, ccsDomainMigration will log an error and exit. Otherwise, ccsDomainMigration will create a lockfile.
5	ccsDomainMigration will use libBeClientIF to connect to the source and destination VWS Voucher and Wallet Server pairs.
6	ccsDomainMigration starts processing the wallets specified in the migration record stored in the SMF database. The migration's state is updated to R in the SMF database and can be viewed from the screens after the data is refreshed (for example by using the **Refresh** button).
7	For each wallet, ccsDomainMigration: • Checks the wallet is on the source VWS using a wallet information request (WI_Req) • Sends a create wallet request (WC_Req) to the destination VWS with a copy of the details and buckets of the wallet • Updates the SMF database by adding new wallet record for the wallet on the destination VWS and deleting the wallet record for the wallet on the source VWS • Sends a delete wallet request (WD_Req) to the source VWS.
8	ccsDomainMigration constructs the migration report and updates the SMF database with the migration status. For more information about the migration report, see *Charging Control Services User's Guide*.
9	ccsDomainMigration removes the lockfile.

Inactive wallet and bucket expiry

This table describes how wallets and buckets are expired due to inactivity.

Note: This is not the same as being expired due to their expiry date being passed.

Step	Action
1	beVWARS loads a wallet. The wallet loaded event triggers *ccsVWARSExpiry* (on page 244). For more information about how beVWARS triggers beVWARS plug-ins, see *Voucher and Wallet Server Technical Guide*.
2	ccsVWARSExpiry checks the wallet state. Go to the appropriate step for the wallet state.
3	If the wallet is currently in the Pre-use state, ccsVWARSExpiry checks the wallet's subscriber batch status. If the batch status is expired, ccsVWARSExpiry sets the wallet status to Terminated.
4	If the wallet is currently in the Active state, ccsVWARSExpiry checks the current date against the wallet's Date Last Used + the Active to Dormant period for the applicable product type. If the current date is later than the wallet's Date Last Used + Active to Dormant period, the wallet is stale. ccsVWARSExpiry: • Writes an EDR detailing the wallet expiry • Sets the wallet state to Dormant For more information about Date Last Used and Active to Dormant, see *Charging Control Services User's Guide*.
5	If the wallet is currently in the Dormant state, ccsVWARSExpiry checks whether the wallet was activated or used. If it was, ccsVWARSExpiry checks the Date Last Used + Active to Dormant period + Dormant to Terminated Period for the applicable product type. If the current date is later than the wallet's Date Last Used + Active to Dormant + Dormant to Terminated, the wallet is stale. ccsVWARSExpiry: • Writes an EDR detailing the wallet termination • Sets the wallet state to Terminated

Expiry event handling

If *ccsVWARSExpiry* (on page 244) is triggered by a wallet expiry event (usually sent by beVWARSExpiry), ccsVWARSExpiry:

- Checks the wallet's expiry date and, if there is none, sets expiry date to now
- Writes an EDR detailing the wallet expiry
- Writes the wallet ID to expired list

The name and location of the expired list is specified by: *expiredPrefix* (on page 169), *expiredSuffix* (on page 247), and *expiredDirectory* (on page 168).
If ccsVWARSExpiry is triggered by a bucket expiry event (usually sent by beVWARSExpiry) and `produceCDRForWalletExpiredBucket` (on page 249) is set to true, ccsVWARSExpiry logs an EDR for the bucket. It does nothing if `produceCDRForWalletExpiredBucket` is false.

If *ccsVWARSPeriodicCharge* (on page 257) is triggered by a bucket expiry event, it processes expiring periodic charge buckets. It keeps the periodic charge bucket and sets the expiry date to a point in the future. For more information about how expiry dates are calculated, see *Charging Control Services User's Guide*.

Wallet removal

This table describes how wallets are removed.

Step	Action
1	beVWARS loads a wallet. The wallet loaded event triggers *ccsVWARSExpiry* (on page 244). For more information about how beVWARS triggers beVWARS plug-ins, see *Voucher and Wallet Server Technical Guide*.
2	If the wallet is currently in the Terminated state, ccsVWARSExpiry checks whether the wallet is passed its wallet expiry date + the Terminated to Removed period for the applicable product type.
3	If the current date is later than the wallet's expiry date + Terminated to Removed, ccsVWARSExpiry checks *logNotRemoveWallet* (on page 248). If logNotRemoveWallet is set to false, ccsVWARSExpiry: • Logs an EDR detailing the wallet removal • Removes all the buckets associated with the wallet • Logs an EDR for each removed bucket • Removes the wallet from the E2BE • The wallet removed event triggers ccsVWARSExpiry again and it logs the wallet removal to the remove list. If logNotRemoveWallet is set to true, ccsVWARSExpiry logs the wallet ID to the remove list. The name and location of the removed list is specified by: *removedPrefix* (on page 169), *removedSuffix* (on page 251), and *removedDirectory* (on page 169). **Exception:** If *removeAtMidnightTZ* (on page 249) is set, ccsVWARSExpiry will take these actions the next time the wallet is loaded after the midnight in the specified timezone which follows the expiry date.
4	If logNotRemoveWallet was set to true, *cmnPushFiles* (on page 277) picks up the remove list from its configured input directory and pushes it to the SMS.
5	cmnReceiveFiles receives the files from cmnPushFiles. For more information about cmnReceiveFiles, see *SMS Technical Guide*.
6	*ccsWalletExpiry* (on page 167) reads files which match the name and location details specified by these parameters: • *removedPrefix* (on page 169) • *removedDirectory* (on page 169).
7	ccsWalletExpiry deletes the wallets from the remove list from the SMF database.
8	ccsWalletExpiry sends a wallet delete request to *ccsBeOrb* (on page 89) for the wallet which was deleted in step 7.
9	*ccsBeOrb* (on page 89) passes the request to beVWARS via beServer.
10	beVWARS attempts to delete the wallet. **Note:** If logNotRemoveWallet was set to false, the wallet will already have been deleted and an error will be returned to ccsWalletExpiry via beServer and ccsBeOrb.

Note: Wallets can also be deleted through the SMS screens. For more information, see *Charging Control Services User's Guide*.

Grace Periods

Wallets can be configured to have a grace state. A grace state provides limited functionality to a wallet which would otherwise be in the terminated state.

A wallet can be in more than one grace period. In this case the functionality is limited to functions allowed by all the applicable grace periods. If a wallet is in more than one grace period, the allowed named events are limited to those events enabled by all the applicable grace periods. Grace periods can only allow named events if the wallet is in Active, Dormant or Terminated states.

Security

Authenticating modules

To provide security over account and voucher generation, CCS contains authentication modules.

These modules contain information uniquely related to the account or voucher number, which is not stored (directly) in the database, but which must be supplied in order to make use of the account or voucher.

Each module has a pair of functions.

1. The first function (the hash generation function) is called at subscriber account- or voucher-generation time.
2. The second (the hash validation function) is called every time a subscriber account- or voucher number is presented to the system during call processing.

Note: Once a batch is created, the authentication module associated with that batch may not be changed.

Modules and security plug-ins

This table describes when security plug-in libraries are used and which authentication module binary they are used by.

Authentication Binary	Use
ccsAccount (on page 297)	Used to generate subscriber account PINs (which are used to secure self-management systems).
ccsVoucherStartup.sh	Used to generate voucher PINs (that is, a string of digits to be printed on the calling card (or similar).
beVWARS ccsVWARSVoucherHandler plug-in	Used to check PIN numbers for validity (for example, to validate a string of digits entered by the user indicating a subscriber account to use or a voucher to redeem).

For more information about the ccsVoucherStartup.sh and ccsVWARSVoucherHandler binaries, see *Voucher Manager Technical Guide*.

Security libraries

Security libraries are used to provide flexibility in how the PINs are generated by *ccsAccount* (on page 297) and ccsVoucher_CCS3. This table describes the function of each security library.

Library	Description
ccsLegacyPIN (on page 142)	Provides the DES authentication rule (DES crypt()ed n-digit PINs) for subscriber account and voucher security. **Note:** The output file is sent directly to the third-party tool gpg, so the resulting printer file is encrypted. The printer file is never created on the SMS in an unencrypted format.

Library	Description
ccsCB10HRNSHA (on page 108)	Provides the CB10 HRN SHA authentication rule for voucher security.
authCB10ValidateSeed *ccsCB10HRNAES* (on page 108)	Provides the CB10 HRN AES authentication rule for voucher security.

Tip: Subscriber account PINs and vouchers are validated using the same security library as they were generated with.

For information about how the authentication rule is selected during:

- Subscriber account generation, see *Charging Control Services User's Guide*
- Voucher generation, see *Voucher Manager User's Guide*

GPG keys

GPG Public keys are used to increase security when creating subscriber account and voucher batch export files for printing.

To use GPG public keys, you must use the Voucher Management screen to:

- Import new GPG public keys
- Verify the imported keys.

Note: You cannot use a key until you verify it.

When a GPG Public Key is imported, it is added to the SMF database by smf_oper. When verified, they are marked as verified. These keys are then available when creating a voucher or account batch. You cannot remove public keys from the database or from the GPG key-ring store on the SMS.

When a voucher batch is created a required key or UID will be supplied. The UID is used to determine which GnuPG key to use within the keyring to encrypt the export file. The key UID is a hexadecimal number up to 10 digits in length.

For more information about the Voucher Management screen, see *Voucher Manager User's Guide*.

Verification of a user-supplied Subscriber Number

The CCS Compatibility Authentication Module is used for subscribers using a PIN. In this case, the CCS Compatibility option is selected from the **Encryption Key** field of the New Subscriber Batch screen or the `-m` option to the batch generation utilities.

The example below illustrates authentication of a subscriber number using subscriber-number-plus-PIN authentication - that is, using the CCS Compatibility authentication module.

Example subscriber account verification

This table shows how a subscriber's account and PIN are verified.

Stage	Description
1	User dials into the gateway.
2	User dials his/her subscriber number and PIN, followed by #.
3	User is presented with a dial tone.
4	User dials destination number.
5	The gatekeeper forwards the subscriber-number/pin and the dialed number to CCS. **Result:** The CCS service logic is invoked.

Stage	Description
6	The subscriber-ID, is looked up in CCS_ACCT_REFERENCE, and the ID of the subscriber-batch is determined. If there is no subscriber-batch for the subscriber, a zero-length hash-digit-string is assumed. Otherwise, the authentication module corresponding to the subscriber-batch is looked up.
7	The subscriber-ID and PIN are sent to the hash validation function, with the private secret retrieved from the CCS_ACCT_REFERENCE row which corresponds to the subscriber's account.
8	If all three pieces of data match, the hash function returns true. In the case of the CCS1 Compatibility security module, it encrypts the secret and compares it to the private secret (which is the PIN encrypted the last time the PIN was set for that subscriber) and returns true if the two encrypted strings match.

Example: The dialed subscriber number and PIN {1033331234 (dialed digit string)} is split into a subscriber-ID (as stored in the database) and a remainder, by using the per service-provider account-number-length parameter.

Note: The TOTAL length of subscriber-ID PLUS ‘secret’ or ‘PIN’ may not exceed 20 digits (for example: 103333 + 1234 (key)+(secret)).

The subscriber-ID, 103333, is looked up in CCS_ACCT_REFERENCE, and the ID of the subscriber-batch is determined. If there is no subscriber-batch for the subscriber, a zero-length hash-digit-string is assumed. Otherwise, the authentication module corresponding to the subscriber-batch is looked up.

At this point, the strings 103333 and 1234 are sent to the hash validation function, along with the private secret retrieved from the appropriate CCS_ACCT_REFERENCE row.

About Secure SSL Connection to the Database

Enabling Secure SSL Connection to the Database

NCC supports secure network logins through Secure Socket Layer (SSL) connections from the NCC UI to the database. SSL is the default method for connecting to the database when you install NCC. You can also enable SSL after installing NCC.

For information about enabling SSL connections to the database, see *SMS Technical Guide*.

Enabling SSL for the CCP

The Customer Care Portal (CCP) provides a customizable user interface (UI) to CCS that allows customer service representatives (CSRs) to perform the tasks required to manage their subscribers.

You can access the CCP through the Services menu in the SMS UI, or you can access it directly from:

- Your Web browser by using the appropriate URL
- A Java WebStart URL
- The desktop or Start menu by using the CCP shortcut

If you access the CCP through the SMS UI and SSL is already enabled, no further action is required to enable SSL for the CCP. For information about enabling SSL on the SMS, see *SMS Technical Guide*.

If you access the CCP directly, enable SSL connections to the database by:

- Creating the Oracle wallet that identifies the database server on the SMS node. Its location must be specified in the **listener.ora** and **sqlnet.ora** files.
- Modifying the **listener.ora** file to additionally listen on port 2484. Use the TCPS protocol for secure SSL connections to the database.

Note: The standard Oracle listener TCP port is 1521. However, SSL connections use the standard port for the TCPS protocol, port 2484, instead. If there is a firewall between screen clients and the SMS, you must open port 2484 in the firewall.

For more information about enabling SSL by configuring the Oracle wallet and updating the **listener.ora** and **sqlnet.ora** files, see *SMS Technical Guide*.

The following additional configuration must be set in the **ccp.jnlp** file:

- The `jnlp.sms.secureConnectionDatabaseHost` Java application property (on non-clustered systems) or the `jnlp.sms.secureConnectionClusterDatabaseHost` Java application property (on clustered systems) must specify the database connection in the CONNECT_DATA part. In addition, the PROTOCOL part must be set to TCPS and the PORT part must be set to 2484.
- If present, set the `jnlp.EncryptedSSLConnection` Java application property to true. The NCC UI connects to the database by using encrypted SSL connections by default.

Note: If you are using non-SSL connections to the database, you must set `jnlp.EncryptedSSLConnection` to false. When `jnlp.EncrtyptedSSLConnection` is set to false, the `jnlp.sms.secureConnectionDatabaseHost` and `jnlp.sms.secureConnectionClusterDatabaseHost` properties are ignored.

See *CCP Application Properties for SSL and Non-SSL Database Connections* (on page 80) for more information.

Calling Card Services

Introduction

The calling card service allows operators to offer a card-based service where a subscriber's calls are charged, not to the CLI or the telephone number of the caller, but to the wallet linked to the subscriber's calling card. The card user dials a predefined service number and security code provided by the telco. This connects them to an IVR system which prompts the caller to enter the destination number to which they wish to transfer the call.

The cost of this call is deducted from the wallet associated with the calling card.

Service features

The calling card service allows the telco operator to:

- Generate large numbers of CCS card/subscriber account numbers randomly in a batch (within the specified range).
- Assign serial numbers to the accounts for customer care purposes.
- Encrypt the output files sent to the print shop and used for producing the printed cards.

Generating account numbers

The ccsAccount command line tool can be used to generate:

- Batches of subscriber/card accounts
- Subscriber/card account PINs (which are used to secure self-management systems)

When the ccsAccount tool is run by ccsAccountWithPrivacy.sh:

- It runs ccsAccount with the `-P` (privacy) parameter
- Account numbers are allocated randomly within the batch, with gaps between the sequences to ensure fraud control (true while the batch is not approaching full)
- A sequential serial number is allocated which is stored in the **CLI** field, while the card number is stored in the **Account Number** field

Note: For more information about ccsAccount, see *ccsAccount* (on page 297).

Setting initial card balance

After the subscriber/card account is generated by ccsAccount, the amount specified in the **Initial Value** field on the New Product Type or the Edit Product Type screen will be credited to the account.

For more information about the Product Type screens, see *Charging Control Services User's Guide*.

Encrypting print shop file

The ccsAccount tool, when run with the `-P` parameter, causes the exported print shop file to be encrypted. The shell script, ccsAccountWithPrivacy.sh, is used to extract the GPG key specified on the command line and directs the encrypted output to the print shop filename.

Example: `ccsAccountWithPrivacy.sh` *`key file ccsAccount_parameters`*

The output is passed onto the ccsAccount binary which then executes with additional parameters:

Example: `ccsAccount -P -m` *`encryption_module ccsAccount_parameters`*

Example

Here is an example ccsAccount command and the resulting account batch output file:

Command:
```
ccsAccount -P -t "World" -m "DES" -s 8815000000 -e 8820990000 -n 10 -
b debit -C 7 -c USD -d 2>&1
```

Output:

```
# Account Batch Output File
# Generated Wed Dec 31 01:24:29 2008
#
AccountBatchID=59
ServiceProviderID=1
AccountTypeID=7
maxConcurrent=1
BatchSize=10
RangeStart=8815000000
RangeEnd=8819990000
AuthenticationModuleID=4
BillingEngineID=2
CurrencyID=2
LimitType=DEBT
BalanceType=1
=
Dec 31 01:24:29.861203 ccsAccount(15179) NOTICE: Beginning account generation.
16309877,3415992,7,G8.H3zCjoKzbY,8800127
19052821,0363266,7,G8fRbQy015unk,8800128
18627603,5447142,7,G82efn9Gh2gSY,8800129
16635167,9003194,7,G8nkF67MOzS9g,8800130
19498256,8441931,7,G8tfZtbQvbOIg,8800131
18758105,8744644,7,G8CSYLULMZtww,8800132
17349265,3517347,7,G8GH/BMl4HHzs,8800133
16223817,0064708,7,G8MbgIe4gPO.U,8800134
16089674,7771756,7,G8lXd7ySSzsVw,8800135
16405822,1207166,7,G8JugOSguxjqg,8800136
Dec 31 01:24:35.514685 ccsAccount(15179) NOTICE: Progress 10/10 (100.0%) Complete
Dec 31 01:24:35.515578 ccsAccount(15179) NOTICE: Account generation complete.
```

Rating and Charging

Introduction

CCS supports different types of charges:

1 Call charging (from the SLC)
2 Named events (from either the SLC or the SMS)

A wallet can also be debited using one of the following:

- A credit transfer (when they pass funds to another wallet)
- A periodic charge (which applies a named event charge on a regular basis)

All charges are calculated and applied by CCS plug-ins on the Voucher and Wallet Servers.

For information about:

- The processing done on the VWS servers, see *Voucher and Wallet Server Technical Guide*.
- How to configure the charges, see *Charging Control Services User's Guide*.

Charging for calls

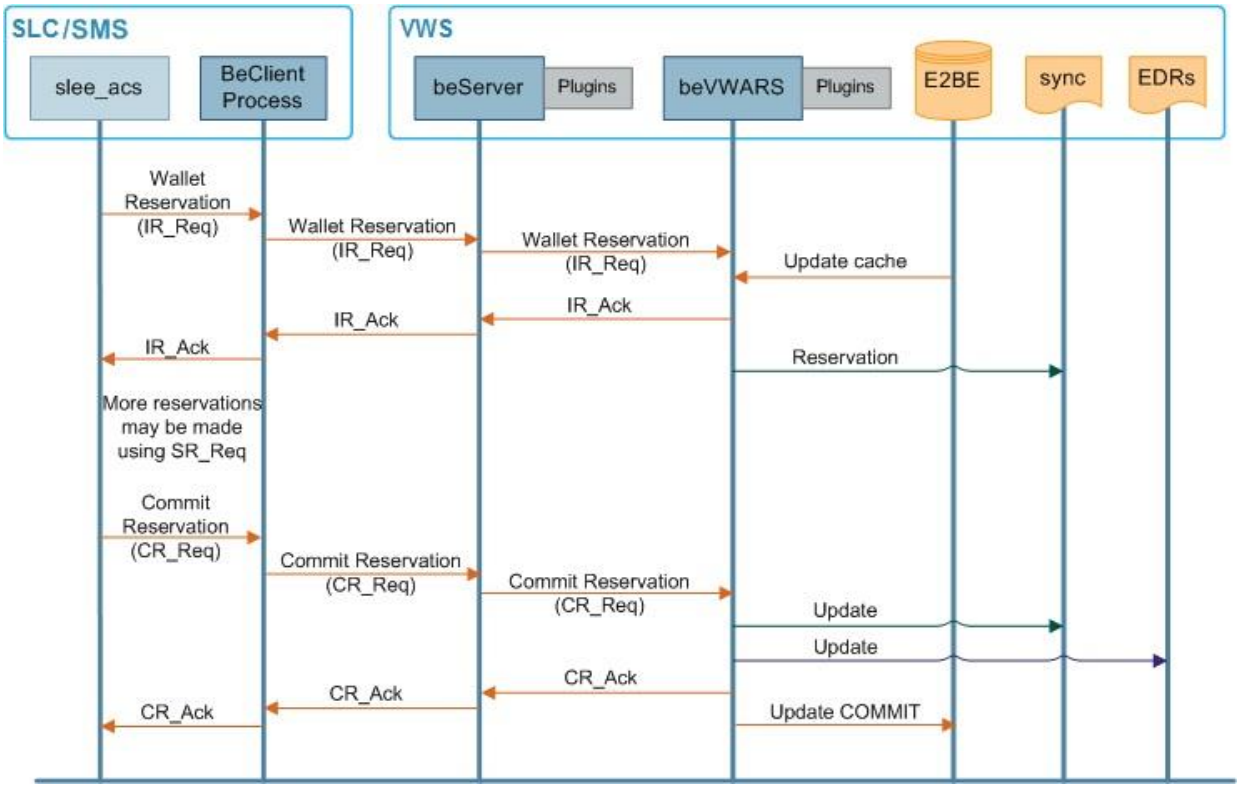
This table describes how CCS handles call rating and charging for a VWS.

Stage	Description
1	Call arrives from network over the SLEE to slee_acs with a service key that triggers the CCS Service Library (*ccsSvcLibrary* (on page 196)). The service to use is determined using the service key, the configuration in the **SLEE.cfg**, and capabilities configuration. For more information about slee_acs, see *Advanced Control Services Technical Guide*.
2	The CCS service library determines the control plan to initiate using the: • Primary wallet of the subscriber's account • Product type of the primary wallet • Capability in the product type that matches the SLEE service key • Control plan matched to the product type capability The control plan which applies to the subscriber is initiated. For more information about configuring capabilities and product types, see *Charging Control Services User's Guide*.
3	Service logic checks for a valid subscriber account to charge by querying beVWARS through BeClient and beServer. **Tips:** • A valid account has a primary wallet. It may also have a secondary wallet. • To use the secondary wallet, you must use the Set Wallet Type feature node in the originating control plan. • The product type's capabilities must be supported by the domain the wallet is on.
4	CCS service library processes the call according to control plan. When the Universal Attempt Billing node is reached, CCS service library sends an Initial Reservation Request (IR_Req) to beVWARS through BeClient and beServer.
5	beVWARS checks for IR message handlers. CCS provides *ccsVWARSReservationHandler* (on page 263) for IR messages, so beVWARS passes the message to that handler. ccsVWARSReservationHandler uses rating tables to calculate the minimum charge to be reserved from a particular balance type to pay for the call. The amount which can be reserved is determined per request, based on:

Stage	Description
	• The balance of the subscriber's account • The value of outstanding reservations • Pending updates. The balances that funds are reserved and charged against are specified in the service's rate table. The rate table can specify more than one balance type by using a balance cascade. **Note:** Reservations may fail due to too many subscribers attempting to access a wallet at the same time.
6	beVWARS checks the wallet. This triggers any beVWARS event plug-ins and they perform any configured actions on the wallet (for details about VWS plug-ins which fire, see *Voucher and Wallet Server Technical Guide*). The only CCS event plug-in which is likely to trigger is *ccsWLCPlugin* (on page 276), which will handle wallets which: • Do not have enough to cover the charge • Have a life cycle period configured If the wallet is still valid, ccsVWARSReservationHandler reserves the charge amount and sends a reservation acknowledgment (IR_Ack) back to the service logic.
Stages 4-6 repeat until the final charge is established by CCS service library. After the first reservation is successfully processed, CCS will use subsequent reservation request (SR_Req) messages to reserve additional blocks of time.	
7	CCS service library finalizes charge (using the Universal Attempt Terminate with Billing node), and sends a commit reservation (CR_Req) request to beVWARS through BeClient and beServer.
8	beVWARS checks for CR message handlers. CCS provides *ccsVWARSReservationHandler* (on page 263) for CR messages, so beVWARS passes the message to that handler. ccsVWARSReservationHandler uses rating tables to calculate the final charge and charges the wallet. **Note:** beVWARS event plug-ins are triggered when the final charge is applied. CCS does not provide any plug-ins which are specifically designed to fire at this point (though ccsWLCPlugin may fire again).
9	beVWARS sends the acknowledgment back to the service logic through beServer and BeClient.
10	The CCS service logic passes the response back to the control plan. If the reservation was successful, the control plan would: • Connect the call. • Continue processing the control plan until an Exit node is reached, then release the call using standard slee_acs release.

Call charging message flow

This diagram shows the message flows involved in charging for a standard voice call.



Charging for Named Events

Named events are predefined events on the system that incur a charge.

This table describes how CCS handles charging for named events for a VWS server.

Stage	Description
1	Named event occurs. **Examples:** • The Named Event feature node is triggered in a control plan. • A periodic charge is triggered. For more information about the Named Event feature node, see *Feature Nodes Reference Guide*.
2	The triggering process (ccsPeriodicCharge on the SMS or slee_acs using the ccsMacroNodes plug-in on the SLC) sends a Named Event (NE) request to the local BeClient process.
3	BeClient process receives the request and sends a NE_Req request to beServer on a Voucher and Wallet Server.
4	beServer on the Voucher and Wallet Server receives the request, calculates the charge, and forwards the request to beVWARS. **Note:** If there are any beServer message handlers configured for NE messages, beServer will pass the request to them before it passes the messages to beVWARS. CCS does not provide beServer message handlers for NE messages described in this process.

Stage	Description
5	beVWARS checks for NE message handlers. CCS provides *ccsVWARSNamedEventHandler* (on page 252) for NE messages, so beVWARS passes the message to that handler. ccsVWARSNamedEventHandler uses Named Event definitions to calculate the named event charge and charges the wallet.
	Note: beVWARS event plug-ins are triggered when the charge is applied. CCS does not provide any plug-ins that are specifically designed to fire at this point (though ccsWLCPlugin may fire).
6	beVWARS sends an acknowledgment back to the service logic through beServer and BeClient.
7	CCS service logic continues processing the control plan until an Exit node is reached, when the call is released using standard slee_acs release.

Note: Named events can also use a reservation process similar to that used in the charging for calls process. In this case three messages are used:

- INER
- SNER
- CNER

For information about how the VWS processes apply the named event charge, see *Voucher and Wallet Server Technical Guide*.

Wallets with multiple concurrent access

Where a wallet has its maximum concurrent accesses field configured to more than 1, charges have special requirements when they are reserved. They can also be applied differently, depending on the application of the `alwaysUsePreferred` parameter.

Terminated State and Wallet Life Cycle periods

Normally, named events and charges cannot be charged against wallets which are pre-use, frozen, suspended, terminated.

However, if a wallet is in a WLC period that allows specific named events, as well as session charges, general charges and general recharges, while being in a terminated state, these will be allowed.

Periodic Charges

Introduction

Periodic charges enable a telco to apply regular charges or recharges to a subscriber's wallet. They can also send notifications on specific events. Periodic charges are configured and populated on the SMS and are run on VWS Voucher and Wallet Servers.

For more information about the configuration available for periodic charges, see *CCS User's Guide*.

Periodic charge processes

This table describes the main processes involved in executing periodic charges.

Process	Role	Further information
beVWARS	Main VWS process. Supports the ccsVWARSPeriodicCharging plug-in and handles interaction with the E2BE database.	*beVWARS* (on page 210)
ccsVWARSPeriodicCharge	This beVWARS plug-in handles periodic charge-specific tasks associated with periodic charge bucket changes.	*ccsVWARSPeriodicCharge* (on page 257)
ccsSLEEChangeDaemon	ccsSLEEChangeDaemon updates assignment of periodic charges to wallets.	*ccsSLEEChangeDaemon* (on page 223)
ccsVWARSWalletHandler	This beVWARS message handler performs the VWS side processing of all messages relating directly to wallets.	*ccsVWARSWalletHandler* (on page 274)
ccsPeriodicCharge	ccsPeriodicCharge applies periodic charges defined for wallets. Only processes periodic charges configured in versions earlier than CCS 3.1.4.	*ccsPeriodicCharge* (on page 144)

Periodic charge processing

This table describes how periodic charges are applied.

Step	Action
1	A wallet is queried. This can be from a normal operation, or because beGroveller passes the wallet ID to beVWARS for groveling. For each bucket that is past its expiry date, an expiry event is generated. For more information about how wallets are groveled, see *Voucher and Wallet Server Technical Guide*.
2	Expiry event triggers *ccsVWARSPeriodicCharge* (on page 257).
3	ccsVWARSPeriodicCharge processes the periodic charge. A periodic charge can apply a charge and/or a credit. According to the periodic charge's configuration, ccsVWARSPeriodicCharge executes: • A named event request (NE_Req), then/or • A wallet general recharge request (WGR_Req for a credit, or VTR_Req for a credit plan (that is, voucher type)). **Note:** Recharges are only applied if the charge was successful. If the debit is unsuccessful, the periodic charge is moved directly to grace or (if the periodic charge has a Loss of Service period of zero) to terminated. EDRs are generated for each operation, unless ccsVWARSPeriodicCharge is processing backlogged charges, in which case an EDR will only be generated if a charge fails and the periodic charge moves to Grace.
4	If the periodic charge should change state (for example, due to a failed charge), ccsVWARSPeriodicCharge: • Applies the state change • Logs an EDR of type 52 For more information about the state transitions and what happens when a periodic charge is applied to a wallet with a disallowed state, see *Charging Control Services User's Guide*.

Periodic charge triggering

The time periodic charges are processed by ccsVWARSPeriodicCharge is based on the following logic:

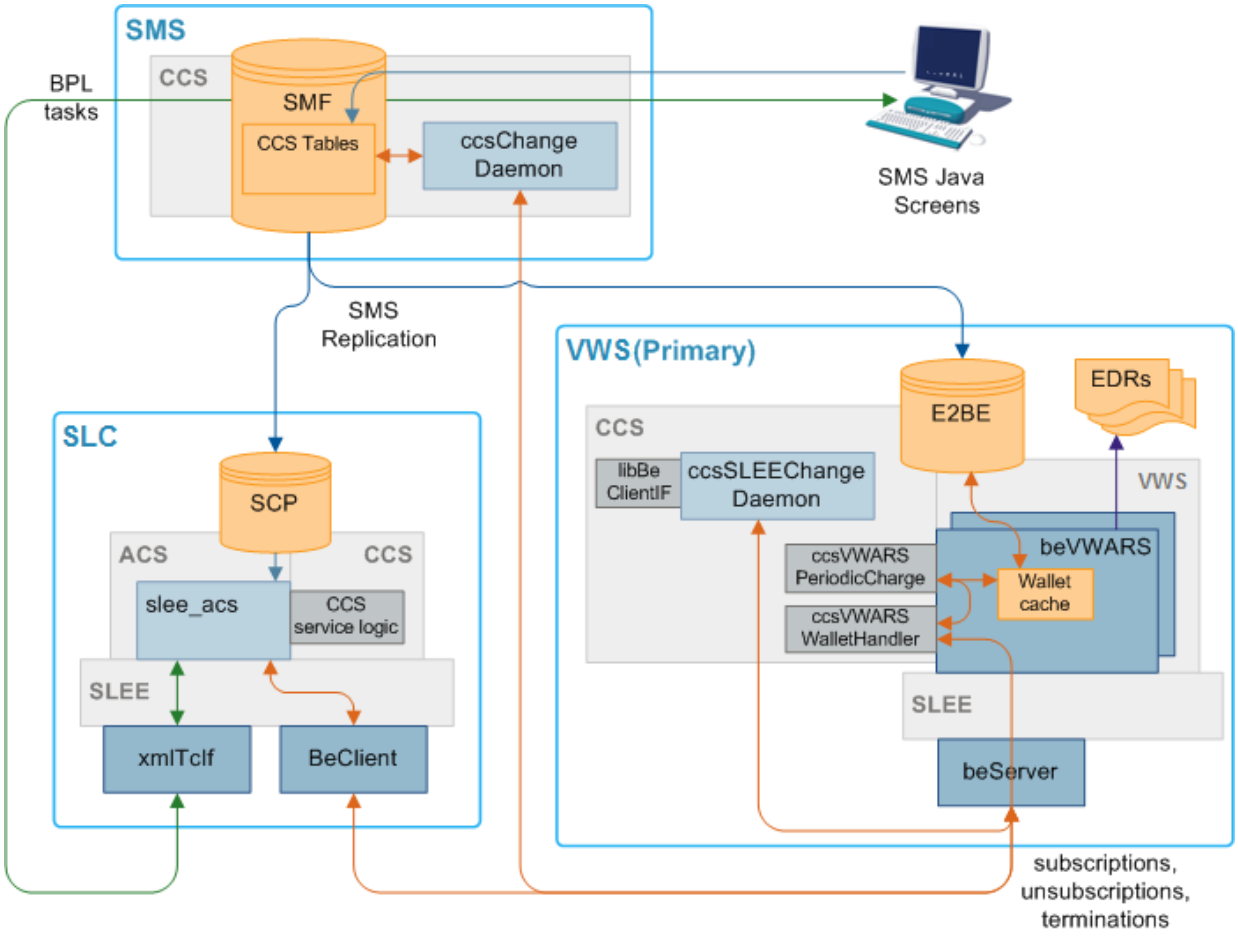
- The periodic charge must have passed its expiry date (this is set based on the details configured in the When option for the periodic charge and where in the periodic charge life cycle the charge is)
  Note: You can adjust when periodic charge processing triggers for a specific time zone by setting the `renewPCAtMidnightTZ` (on page 251) parameter in the ccsVWARSExpiry section of the **eserv.config** file.
- The wallet must have been queried (either from normal activity, or because beVWARS's groveller processed the wallet from work sent from beGroveller)
- For fixed date charges, the value set in *chargeTimeGMTHours* (on page 259)
- The processing of the wallet can be delayed by *retryTimeoutMinutes* (on page 261)

For more information about:

- 'When' configuration for a periodic charge and the periodic charge life cycle, see *Charging Control Services User's Guide*.
- When the beGroveller will send a wallet to be groveled by beVWARS, see *Voucher and Wallet Server Technical Guide*.

Periodic charge association maintenance diagram

This diagram shows how periodic charge to wallet associations are maintained.



Processing Periodic Charge Subscription Changes

This table describes how changes to periodic charge states are processed.

Step	Action
1	Periodic charge subscriptions are triggered when: • A customer service representative or subscriber triggers a periodic charge subscribe, unsubscribe or terminate BPL task using the Periodic Charge Subscription feature node. • A customer service representative or subscriber triggers a periodic charge transfer using the Periodic Charge Transfer feature node in a control plan. • A periodic charge configuration change is made through the SMS screens (*ccsSLEEChangeDaemon* (on page 223) or *ccsVWARSActivation* (on page 241) sends WU_Req with state change (see *Periodic charge assignment* (on page 35) for more information) to beVWARS). • *ccsVWARSPeriodicCharge* (on page 257) calculates and applies a final charge.
2	If the trigger is a periodic charge subscription, unsubscription or termination of a subscription to a service, a wallet update request (WU_Req) is sent from the BPL control plan's Periodic Charge Subscription feature node with the: • Subscriber's ID • Change value (that is, Subscribe (103), Unsubscribe (102), or Terminate (101)) • Periodic charge ID For more information about BPL tasks, see the *Task Management* chapter in *Charging Control Services User's Guide*. For more information about the Periodic Charge Subscription feature node, see *Feature Nodes Reference Guide*. If the trigger is a periodic charge transfer, a wallet information query (WI_Req) is completed against the subscriber's wallet. The query returns information about the subscriber's current subscription balances. If the subscriber has a subscription which is not in an Unsubscribed or Terminated state, the Periodic Charge Transfer feature node sends a wallet update request (WU_Req): • Changing the existing subscription balance to terminated • Creating a new subscription balance and buckets for the target periodic charge (copying the expiry date to the new balance).
3	The WU_Req is received by *beVWARS* (on page 210) on the VWS server and *ccsVWARSWalletHandler* (on page 274) is triggered. When ccsVWARSWalletHandler receives a periodic charge subscription request (WU_Req 103), it checks for the presence of a periodic charge balance type for this periodic charge in the wallet (that is, whether the periodic charge is assigned to the subscriber's product type). If the wallet does not have the relevant periodic charge balance type, ccsVWARSWalletHandler creates the balance type which correlates to the periodic charge ID sent in the WU_Req and creates a bucket for the new subscription with an initial value of 103. If the request is unsubscribe or terminate (WU_Req 102 or 101), and the required balance type does not exist, ccsVWARSWalletHandler returns a Not Subscribed error. The WU_Reqs from the periodic charge transfer are treated as normal balance updates. **Note:** The EXPIRY value is not changed. If the expiry has been changed by a WU request (in error), then it will be reset back to the original EXPIRY value before applying the state machine logic.

Step	Action
4	ccsVWARSWalletHandler triggers bucket and/or a balance value changed events as necessary to reflect changes. Exception: If the bucket or balance value is due to a periodic charge transfer, ccsVWARSWalletHandler does not trigger a bucket and/or balance changed event (and step 5 and 6 are skipped). **Note:** If no action is described in step 3, the balance type change event is the only action ccsVWARSWalletHandler will take.
5	Any bucket or balance changed event triggers the *ccsVWARSPeriodicCharge* (on page 257) plug-in. **Note:** ccsVWARSPeriodicCharge is triggered on all bucket or balance changed events, but only processes periodic charge balances.
6	ccsVWARSPeriodicCharge checks for periodic charge balances and buckets. For periodic charge balances and buckets, ccsVWARSPeriodicCharge: • Changes the state value to reflect the new state (that is, subscribed, unsubscribed or terminated) • Recalculates and updates the bucket's expiry date • Triggers any configured notifications For more information about configuring periodic charge expiries and notifications, see *Charging Control Services User's Guide*.

Periodic charge assignment

This table describes how periodic charge to wallet relationships are updated.

Step	Action
1	The periodic charge is configured on the SMS screens and is saved to the SMF database.
2	When a periodic charge is changed so it is assigned to a product type and 'Apply to Existing' is selected, the change to the CCS_AT_PERIODIC_CHARGE table triggers adding a new record to CCS_PC_QUEUE. This change is also replicated to the E2BE database on the VWS using SMS replication. **Note:** If the periodic charge has 'Apply to Activating Subscribers' selected, an entry is also added to CCS_PROMOTION, and the relationship is handled by ccsVWARSActivation. For more information, about this process, see *Periodic charges and wallet activation* (on page 36).
3	*ccsChangeDaemon* (on page 131) on SMS and *ccsSLEEChangeDaemon* (on page 223) on VWS polls the CCS_PC_QUEUE table and picks up the new record. **Note:** Polling frequency is controlled by pollPeriod. The frequency records are processed at is controlled by *throttle* (on page 133).
4	If the CCS_PC_QUEUE record has a change type of A (that is, a periodic charge has been associated with or removed from a product type), ccsSLEEChangeDaemon on the VWS sends a wallet inquiry request (WI_Req) to check subscriber's subscription status.

Step	Action
	Note: This query will be processed as a normal WI_Req on the VWS VWS. That is, it will trigger the WI message handler, and any event plug-ins which are triggered by wallet query events. For more information about event plug-ins, see *Background Processes on the VWS* (on page 209).
	• If the change action = I, and the wallet inquiry reports the balance type and bucket do not exist or they do exist but are set to Terminated, sends *beVWARS* (on page 210) a wallet update request (WU_Req) which sets the periodic charge's state to subscribed. • If the change action = D, and the wallet inquiry reports the balance type and bucket for this subscriber exist and are not set to Terminated, sends beVWARS a wallet update request (WU_Req) which sets the periodic charge's state to terminated.
5	If the CCS_PC_QUEUE record has a change type of W (that is, a single wallet has been associated with a periodic charge), ccsChangeDaemon on the SMS loops through each periodic charge. For each periodic charge which is associated with the wallet's product type and has "marked as apply to existing subscribers": • If the change action = I (association), ccsChangeDaemon sends beVWARS a wallet update request (WU_Req) which sets the periodic charge's state to Subscribed. • If the change action = D (removal), ccsChangeDaemon sends beVWARS a wallet update request (WU_Req) which sets the periodic charge's state to Terminated.
6	If the CCS_PC_QUEUE record has a change type P (that is, a wallet has swapped product types), ccsChangeDaemon on the SMS loops through the wallet's periodic charges checking for periodic charges that are no longer relevant and for new periodic charges from the new product type being swapped to. • For the periodic charges associated with the old product type and not associated with the new product type, ccsChangeDaemon sends beVWARS a wallet update request (WU_Req) which sets the periodic charge's state to Terminated. • For the periodic charges associated with both the old and the new product types the ccsChangeDaemon does nothing, regardless of the state of the subscription to that periodic charge. • For the periodic charges which are associated with the new product and "marked as apply to existing subscribers" and for which the subscriber has no subscription, ccsChangeDaemon sends beVWARS a wallet update request (WU_Req) which sets the periodic charge's state to subscription.
7	When ccsSLEEChangeDaemon receives confirmation of the update, it removes the CCS_PC_QUEUE record.

Periodic charges and wallet activation

In addition to the operations normally performed when a subscriber's subscription to a periodic charge changes, operations may be performed when a subscriber:

- Activates a wallet or resubscribes when their periodic charge is in a terminated state
- One or more of the periodic charges associated with the wallet's product type have 'Apply to Activating Subscribers' ticked

If the change is a wallet state change from PreUse to Active, *ccsVWARSActivation* (on page 241) applies any activation credits (CCS_PROMOTION entries) as per standard behavior. For any periodic charge which has 'Apply to Activating Subscribers' ticked, an activation credit is defined which includes the periodic charge's balance type and a bonus which has a value of 103 (subscribe). When the credit is applied and ccsVWARSActivation attempts to created the relevant subscription bucket, *ccsVWARSPeriodicCharge* (on page 257) is triggered and creates the appropriate periodic charge balance in the wallet.

Note: When a periodic charge is subscribed-to an immediate charge (Named Event) is not taken (unless one is specified in the control plan executed by the BPL task which changes the subscriber's periodic charge state. This enables any issues with sequencing of activation credits to be avoided.

If a wallet state is changed from Terminated to Active, *ccsVWARSPeriodicCharge* (on page 257) searches for periodic charges in Terminated state. Any periodic charges that are configured to ‘Apply to Activating Subscribers’ are changed to Subscribed. Any other periodic charges are left in the Terminated state.

For more information about 'Apply to Activating Subscribers' field, see *Charging Control Services User's Guide*.

Sending periodic charge notifications

This table describes how notifications generated by periodic charges are sent.

Step	Action
1	When *ccsVWARSPeriodicCharge* (on page 257) executes a transition which sends a notification, it writes a notification request to the notification batch file. **Exception:** No notifications will be sent if either: • ccsVWARSPeriodicCharge is processing backlogged PreCharge transitions • The state of the affected wallet is not allowed The time the notification is written is controlled by *notificationMidnightTZ* (on page 260). For more information about which transitions send notifications and how to configure them, see *Charging Control Services User's Guide*.
2	From there, the standard real-time notifications subsystem processes the notifications as usual. For more information about how real-time notifications are processed, see step 3 in the Real-time wallet notifications process.

Recharges

Recharge methods

CCS supports either off-the-shelf or customized recharge mechanisms depending on which interfaces are available. This table describes the available recharge mechanisms.

Recharge method	Description
Voucher / Scratch Card recharge	A voucher creation, management and replenishment system is provided with the VWS which a subscriber can use to recharge their wallets. Vouchers can be redeemed using any of the following interfaces: • IVR interaction • USSD interaction • PI-integrated web portals

Recharge method	Description
SMS GUI	Telco operators can recharge subscriber accounts using the SMS administration screens: • **Free Form Recharge** tab on the Wallet Management screen • **Voucher Recharge** tab on the Voucher Management screen
Credit Card Recharge	Prepaid Charging stores credit card information so a subscriber can be recharged against a credit card number previously provided by the subscriber (when authorized by PIN entry). Credit cards can also be charged periodically (for example, one account charge per month).
Web	The PI can support command execution from a range of sources (for example: websites).
Electronic refill	Systems have been deployed that use ISO 8583-based interfaces to recharge subscriber accounts directly from: • Bank accounts • ATMs • Other banking mechanisms

Tip: Wallets can also have credit added as part of a promotion or bonus.

Subscriber interaction

CCS handles recharges by using subscriber interaction:

- IVR feature nodes in a control plan
- Customer care service staff using SMS screens
- (with MM) Short Messages
- (with USSD GW) menus and fast access

Promotions

Introduction

Promotions can be used to increase subscriber activity by rewarding subscribers with more attractive packages for specific behavior. Promotional bonuses can be implemented using one of the following:

- In-built rewards and bonus schemes configured in Promotion Manager
- Free form configuration such as control plans and/or profile fields

In-built reward and bonus types

This table describes the types of in-built rewards and bonuses provided to CCS through Promotion Manager.

Type	Description
Tracker threshold promotions	Awarded to subscribers whose total usage exceeds a set threshold. Promotional reward can change the subscriber's product type (and applicable tariff), and/or award one or more bonus credits. Promotion notifications can be sent to subscribers specifying how much more they need to spend to upgrade.

Type	Description
Wallet activation promotions	Triggered when a subscriber activates their account. Defines a time period from subscriber creation to activation. If a subscriber activates their account in this period, they are given free SMS messages.
Balance recharge promotions	Awards a promotional cash bonus to subscribers if they recharge their account and the recharge is above a specified threshold.

Promotions process

Balance changes due to promotions are handled by the ccsPMXPlugin on the VWS. For details, see *ccsPMXPlugin* (on page 238).

More information

Promotions are configured in Promotion Manager. For more information about promotions, see *Promotion Manager User's Guide*.

Notifications

Introduction

Notifications are any short message sent by CCS to a subscriber's handset.

CCS sets up notifications which are delivered by other applications. Different delivery applications are used depending on the type of network and destination.

ACS Notification Templates

You define the content to include in notifications by configuring ACS notification templates. For more information, see *ACS User's Guide*.

Examples of CCS activities that can use ACS notification templates are:

- Feature nodes in control plans
- Business process logic (BPL) tasks
- Credit transfers
- Periodic charges
- Profile updates
- Real-time notifications
- Promotions

Notification Languages

Notifications can use any language configured on the system. They are sent in the subscriber's preferred language (if set) or in the system's default language.

For more information about configuring:

- Languages, see *ACS User's Guide*
- Notification translations, see *CCS User's Guide*

Events Triggering Notifications

This table lists the events triggering notifications sent by CCS.

Notification	Triggering Events	Delivery by
Control plan notifications	Requested by a feature node in a control plan; for example, to send: • Account Status SMS • Call Information SMS • SMS Low Balance **Note:** This includes control plans used by BPL tasks.	Notifications DAP template
Real-time wallet notifications	A specific change in wallet and balance details on the VWS, including: • Balance or wallet expiry warning • Balance charge • Balance recharge • Wallet state change Promotions, including: • Heavy user rewards	Notifications DAP template
Periodic charge notifications	Successful or unsuccessful periodic charges	Notifications
CCS System notifications	A specific event in CCS including: • Periodic charge success or failure • Entry to, or exit from, a wallet grace period	Notifications DAP template
Credit Transfer notifications	Credit transfer success or failure	Notifications
Profile notifications	A defined event in a subscriber's profile	Notifications DAP template

For more information about:

- ACS notifications, see *ACS User's Guide*
- DAP templates, see *DAP User's Guide*
- Profile notifications, see *Charging Control Services User's Guide*

About Notification Delivery

Notifications can be delivered by:

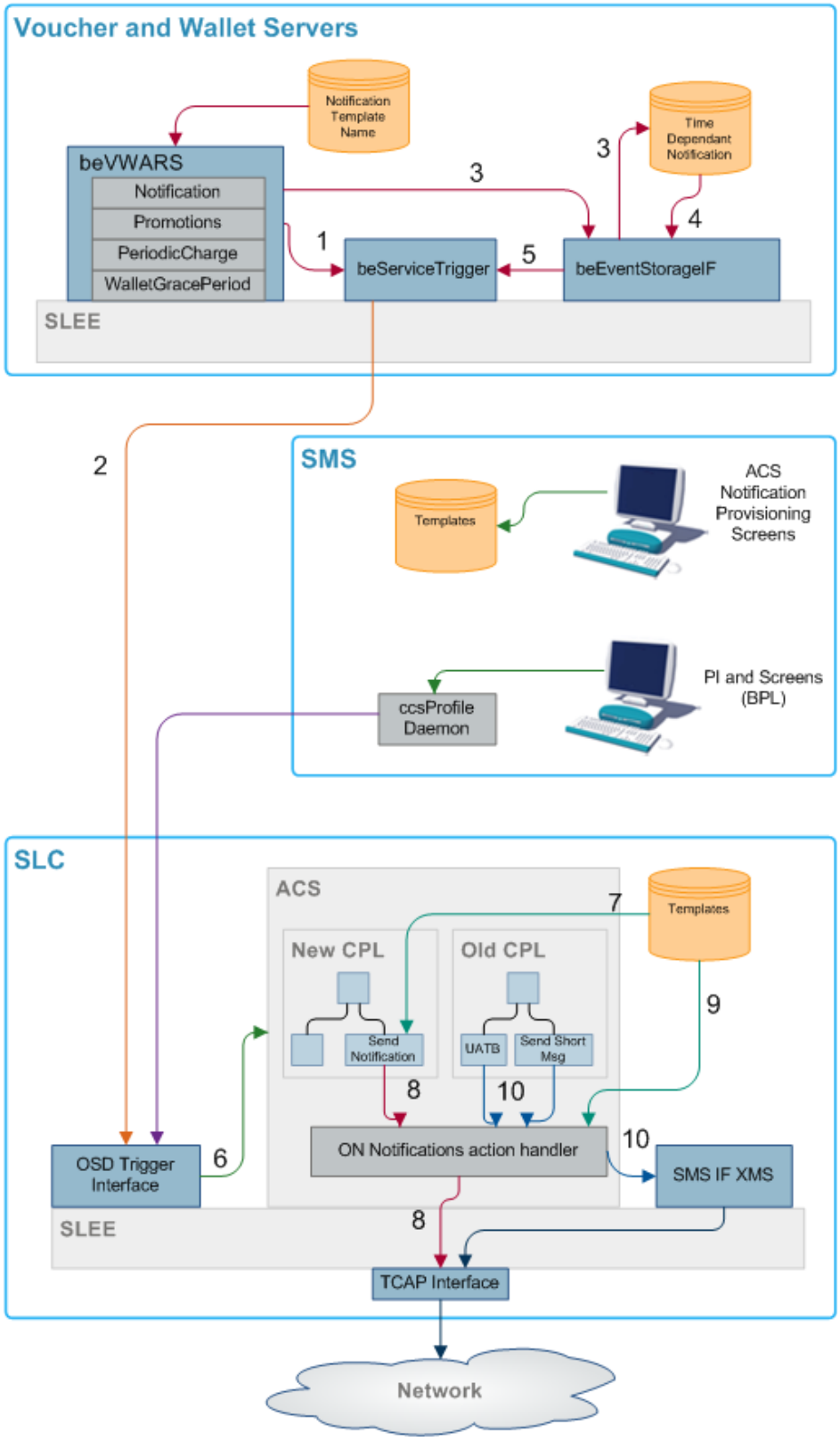
- slee_acs process (called by feature nodes in control plans)
- SMSC IF (smsInterface)
- Messaging Manager (xmsTrigger)
- The ccsProfileDaemon or xmlIF processes (through DAP XML templates)

For more information about:

- smsInterface, see *SMSC Technical Guide*
- xmsTrigger, see *MM Technical Guide*
- DAP XML templates, see *Data Access Pack User's & Technical Guide*

Notification Flows

This diagram shows the various notification flows across the NCC platform.



Flow 1

The beVWARS plug-ins send SMS information to the beServiceTrigger.

Flow 2

Notification XML messages from the beServiceTrigger to the OSD interface on the SLC.

Flow 3

If a notification cannot be delivered immediately, either because it has an associated time period when it can be delivered, or because the delivery attempt failed, then persistent storage of the notification is provided in a database table.

Flow 4

The beEventStorageIF process looks for, and retrieves, the notification entries in the database that can be sent now, either because their allowable delivery time has been met, or because the notification is a message retry.

Flow 5

The beEventStorageIF deletes the active notification entries from the database and sends delivery request messages to the beServiceTrigger for each one.

Flow 6

The OSD interface triggers ACS, which then loads the control plan containing the notification feature node that will perform delivery of the notification.

Flow 7

The notification template to use is determined by the notification feature node, based on:

- Language ID
- Template ID
- Customer ID

Flow 8

The notification feature node delivers a USSD notification through the TCAP interface.

If the message class is "USSD push", then an internal message is sent through the USSD push action handler to the TCAP interface after the notification feature node has performed all the parameter substitutions.

Flow 9

Chassis action to construct message from template.

Flow 10

Other send message feature nodes use new chassis actions to deliver notifications using Messaging Manager.

EDRs

Introduction

This topic explains how EDRs are used in CCS. Most of the information relates to processing of the EDRs after they are written. For more information about how EDRs are generated, see *VWS Technical Guide* and *Event Detail Record Reference Guide*.

Viewing active rules for a subscriber

Follow these steps to view the active rules for a subscriber.

Step	Action
1	Open the Subscriber Management screen for the Prepaid Charging service.
2	On the **Subscriber** tab, select the subscriber record and click **Edit**.
3	In the left pane of the Edit Subscriber screen, select the **Balance Topup Rules** option. **Result:** The Balance Topup Rules screen appears. The rules that apply to this subscriber are displayed on the screen. You see the name of the rule and the date for the last time it will be executed. **Note:** This information is read only.

Dataflow

This table shows the process by which EDRs are written and collected to the SMF database.

Stage	Description
1	The SLC is the originator of all events that cause Voucher and Wallet Servers to perform tasks during call processing, as the SLC controls how the service responds to network events. The SLC signals events to the VWS Voucher and Wallet Server using the CCS Billing Engine Protocol. The service sends messages to the Voucher and Wallet Servers through the ccsBeClient interface.
2	EDRs are written out to disk as ASCII files on the VWS.
3	The files are transfered to the SMS.
4	The files are indexed and made available to the Java User Screens and external EDR post-processing tools.
5	CCS screens created EDRs are written by the ccsCDRGenerator process to the same directory the VWS flat files are transfered into. The ccsCDRLoader then loads both the same way.

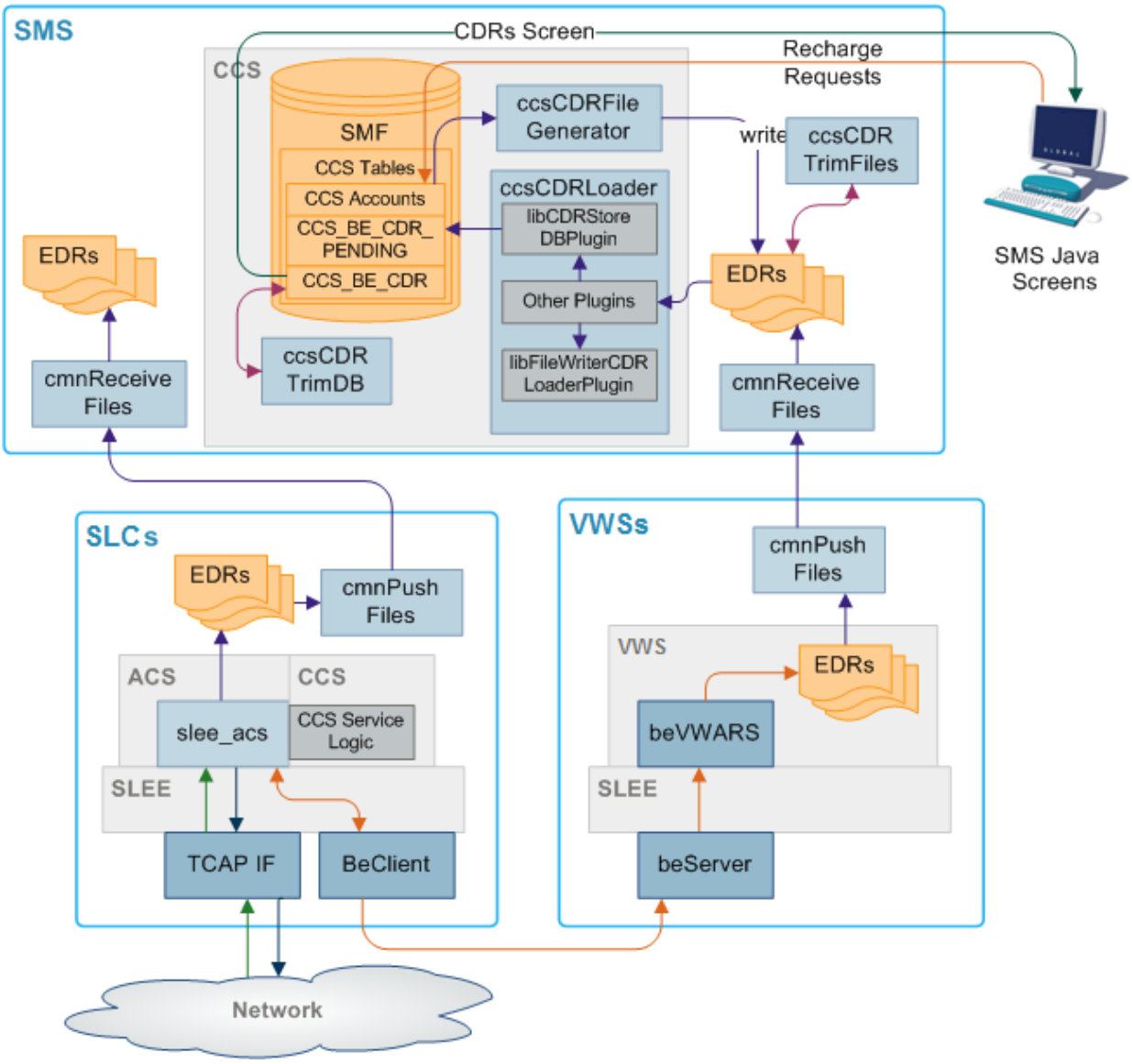
CCS EDR processing

This process shows how EDRs are processed on the SMS by CCS components.

Step	Action
1	If configured to, ccsCDRTrimFiles processes the EDRs from the VWS.
2	ccsCDRLoader inserts the details from the EDR files into the CCS_BE_CDR table in the SMF database.
3	If configured to, ccsCDRTrimDB processes the EDRs.
4	EDRs can be viewed on the EDR Details screen in CCS.

Diagram

Here is an example showing EDR creation, transfer to the SMS and processing.



Process descriptions

This table describes the processes involved in Voucher and Wallet Server EDR creation, transfer and processing in CCS.

Note: EDRs are also created on the SLC to record the details of the call processing through the control plan and slee_acs.

Process	Role	Further information
beVWARS	beVWARS writes EDRs on the VWS based on VWS account, wallet and balance transactions.	*VWS Technical Guide*
cmnPushFiles	cmnPushFiles reads EDRs on the VWS and sends them to a configured directory on the SMS. Once the files have been sent, the read files on the VWS are archived by cmnPushFiles.	*cmnPushFiles* (on page 277)

Process	Role	Further information
cmnReceiveFiles	cmnReceiveFiles accepts EDRs sent from cmnPushFiles and writes them to the directory on the SMS specified by cmnReceiveFiles.	*SMS Technical Guide*
ccsCDRLoader	ccsCDRLoader scans the input directory written to by cmnReceiveFiles and loads any EDRs into the CCS_BE_CDRS table in the SMF database.	*ccsCDRLoader* (on page 111)
ccsCDRFileGenerator	ccsCDRFileGenerator creates EDRs recording relevant actions taken in the CCS UI screens. Relevant actions include changes to the balances or wallets.	*ccsCDRFileGenerator* (on page 108)
ccsCDRTrimDB	ccsCDRTrimDB periodically scans the CCS_BE_CDR table in the SMF and removes records past a specified age.	*ccsCDRTrimDB* (on page 129)
ccsCDRTrimFiles	ccsCDRTrimFiles periodically scans the EDR archive directory on the SMS and removes files over a specified age.	*ccsCDRTrimFiles* (on page 130)
CCS UI screens	The CCS screens enable: • Subscriber details and wallets to be updated through EDRs created by ccsCDRGenerator • EDRs in CCS_BE_CDR to be viewed	*Charging Control Services User's Guide*

EDR triggers

EDRs are written on the Voucher and Wallet Servers when a wallet or voucher is modified. The following messages, among others, cause the beVWARS to write EDRs:

- Call End Notification
- Wallet Recharge Request
- Named Event

Chapter 2

Configuration

Overview

Introduction

This chapter explains how to configure the Oracle Communications Network Charging and Control (NCC) application.

In this chapter

This chapter contains the following topics.

Configuration Overview 47
Configuring the Environment 49
eserv.config Configuration 50
Configuring acs.conf for the SLC 53
Setting up the Screens 60
Defining the Screen Language 64
Defining the Help Screen Language 65
Configuration Through the ACS Screens 66
User Interface-Based Configuration Tasks 67
Configuring VWS processes for CCS 68
Configuring CCS Macro Nodes 69
Switch Configuration for the UATB Node 71
Voucher Status Report Configuration 73
CCP Configuration 74

Configuration Overview

Introduction

This topic provides a high level overview of how the CCS application is configured.

There are configuration options which are added to the configuration files that are not explained in this chapter. These configuration options are required by the application and should not be changed.

Configuration process overview

This table describes the steps involved in configuring CCS for the first time.

Stage	Description
1	The environment CCS will run in must be configured correctly. This includes: • If a the directory CCS was installed into was not the recommended directory, setting the root directory • Setting the Oracle variables • Configuring the location of the EDR directories • Configuring the ccs_oper profile • Configuring the web server • Configuring CCS Balance Top Up Suite
2	The **eserv.config** file must be configured for CCS. The example file should be copied into the main **eserv.config**, and any required details configured. For more information, see **eserv.config** *Configuration* (on page 50).
3	**acs.conf** must be configured to include CCS on all SMSs and SLCs.
4	If the default language for the CCS graphical user interface need changing, the new default language must be configured.
5	The CCS screen-based configuration must be completed.
6	If the VWS has been installed, the VWS processes must be configured.

Configuration components

CCS is configured by the following components:

Component	Locations	Description	Further Information
eserv.config	All machines	The most important is **eserv.config**, because it configures most NCC applications, including the VWS processes used by CCS. CCS is configured by the CCS section of **eserv.config**.	**eserv.config** *Configuration* (on page 50).
acs.conf	All SMS and SLC nodes	The **acs.conf** file configures the: • acsChassis which processes calls on the SLC • acsCompilerDaemon which compiles control plans, geography trees and CLI-DN files on the SMS.	*Configuring acs.conf for the SLC* (on page 53) *Advanced Control Services Technical Guide*
CCS UI	SMF database	The CCS UI allows you to configure many parts of CCS.	*User Interface-Based Configuration Tasks* (on page 67) and *Charging Control Services User's Guide*
	SLC nodes		*Voucher Status Report Configuration* (on page 73)

Configuring the Environment

Oracle variables

The CCS Unix system accounts ccs_oper and ebe_oper require the standard ORACLE environment variables to be present.

Configuring EDR log directories

Because most systems will generate a large number of EDRs, it is recommended that the EDR log directories are changed from the default install values.

A link should be created between the default logging directories and the actual location on separate physical disk, apart from the main application installations.

You must create links from the following directory on the VWS:

- **/IN/service_packages/BE/logs/CDR**

You must create links from the following directories on the SMS:

- **/IN/service_packages/CCS/logs/CDR**
- **/IN/service_packages/CCS/sync/tmp**

Procedure

Follow these steps to configure the location of the EDR log directories.

Note: These steps assume /volD is the mount point for the disk that EDRs are to be stored on.

Step	Action
1	Change to the volume where the EDRs should be kept. **Example command:** `cd /volD`
2	Create a EDR directory. **Example command:** `mkdir CDR` **Result:** This creates the EDR directory.
3	Change to the CCS log directory. **Example command:** `cd /IN/service_packages/CCS/logs`
4	Move the EDR directory's contents to the EDR directory on the alternative volume. **Example command:** `mv CDR/* /volD/CDR` **Note:** The move command may fail, if so repeat.
5	Delete the EDR directory. **Example command:** `rmdir CDR`
6	Create a link from the application's EDR directory to the new EDR directory on the alternative volume. **Example command:** `ls -s /volD/CDR /IN/service_packages/CCS/logs/CDR` **Result:** This links the new location to the old name. CCS will write all EDRs to the new location.

Configuring the .profile

If ACS and CCS are installed, follow these steps to edit the **.profile** file to set the path correctly.

Step	Action
1	Open the .profile file for editing. **Example command:** `vi <ACS_ROOT>/.profile-scp`
2	Add the following line: `export LD_LIBRARY_PATH=<CCS_ROOT>/lib:$LD_LIBRARY_PATH`
3	Save and close the file.

Configuring CCS Balance Top Up Suite

The `UTL_FILE_DIR` parameter defines the directories the utl_file package, used by CCS Balance Top Up Suite, needs for writing files. You must add this parameter to the **initSMF.ora** file.

Procedure - adding UTL_FILE_DIR

Follow these steps to add the `UTL_FILE_DIR` parameter to the **initSMF.ora** file. This enables access to the file system.

Step	Action
1	Log in to the SMF server as the Oracle unix user: Type `su - oracle` *password*
2	Locate the oracle parameter file **initSMF.ora** in the **$ORACLE_BASE/admin/SMF/pfile/** directory**.**
3	Add both the following `UTL_FILE_DIR` parameters to **initSMF.ora** on the SMF server: `UTL_FILE_DIR=/IN/service_packages/CCS/tmp` `UTL_FILE_DIR=/IN/service_packages/CCS/tmp` **Result:** The utl_file package now has access to the file system.
4	Restart the SMF Oracle instance.

eserv.config Configuration

Introduction

The **eserv.config** file is a shared configuration file, from which many Oracle Communications Network Charging and Control (NCC) applications read their configuration. Each NCC machine (SMS, SLC, and VWS) has its own version of this configuration file, containing configuration relevant to that machine. The **eserv.config** file contains different sections; each application reads the sections of the file that contains data relevant to it.

The **eserv.config** file is located in the **/IN/service_packages/** directory.

The **eserv.config** file format uses hierarchical groupings, and most applications make use of this to divide the options into logical groupings.

Example eserv.config detail

This configuration sample shows an example of a part of an **eserv.config** file showing a CCS wallet handler:

```
CCS = {
    reservationHandler = {
```

```
        reservationLengthTolerance = 60 # in milliseconds
    }
}
```

Configuration File Format

To organize the configuration data within the **eserv.config** file, some sections are nested within other sections. Configuration details are opened and closed using either { } or [ ].

- Groups of parameters are enclosed with curly brackets – { }
- An array of parameters is enclosed in square brackets – [ ]
- Comments are prefaced with a # at the beginning of the line

To list things within a group or an array, elements must be separated by at least one comma or at least one line break. Any of the following formats can be used, as in this example:

```
    { name="route6", id = 3, prefixes = [ "00000148", "0000473"] }
    { name="route7", id = 4, prefixes = [ "000001049" ] }
```

or

```
    { name="route6"
        id = 3
        prefixes = [
            "00000148"
            "0000473"
        ]
    }
    { name="route7"
        id = 4
        prefixes = [
            "000001049"
        ]
    }
```

or

```
    { name="route6"
        id = 3
        prefixes = [ "00000148", "0000473" ]
    }
    { name="route7", id = 4
        prefixes = [ "000001049" ]
    }
```

eserv.config Files Delivered

Most applications come with an example **eserv.config** configuration in a file called **eserv.config.example** in the root of the application directory, for example, **/IN/service_packages/eserv.config.example**.

CCS eserv.config example file

CCS delivers a cut-down **eserv.config** file that only contains non-default parameters; it is not a full list of all parameters that are available. This file will normally be installed as **eserv.config**, except in the case that another application has already installed **eserv.config**.

Some specific parameters (for example host names) will need to be amended in the installed **eserv.config** file; these are clearly marked with "Change Me" markers. Once amended, CCS will run with no further changes to **eserv.config**. Where additional implementation changes need to be made to **eserv.config**, refer to the *Background Processes* chapters for full descriptions of all parameters for the processes.

In addition, a full example file containing examples of all parameters and parameter descriptions is also delivered. This example file is called **eserv.config.ccs_example**.

Parameters

Listed below are the parameters in the `CCS` section that are common to all machines.

accountNumberLength

Syntax:	`accountNumberLength = int`
Description:	The number of digits in card number in a subscriber account. If accountNumberLength is set to zero (0) then the account number can be any length.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	Used by ccsAccount when generating subscriber accounts.
Example:	`accountNumberLength = 14`

oracleUserAndPassword

Syntax:	`oracleUserAndPassword = "user/pwd[@db_sid]|/@connection_string"`
Description:	The user credentials that CCS uses for connections to the database on a local or remote SMS node when the `oracleUser` or `oraclePassword` parameters are not defined.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	For connections to a: • Local database, specify the user and password, or specify '/' for passwordless connections • Remote database, specify the user, password and database SID • Local or a remote database by using the Oracle wallet secure external password store, specify only the TNS connection string where the TNS connection string is the alias defined for the username and password credentials in the external password store. This alias can be either a TNS name or a service name from **tnsnames.ora**.
Default:	/
Notes:	You can specify the user credentials for connecting to the database in the `oracleUser` or `oraclePassword` parameters for some CCS processes. In this case, the `oracleUserAndPassword` parameter is ignored.
Example:	`oracleUserAndPassword = "smf/smf"`

suppressedEDRTags

Syntax:	`suppressedEDRTags = ["EDRtags"]`
Description:	Some EDR tags can be optionally hidden when creating an EDR.
Type:	Array
Optionality:	Optional
Allowed:	Optional tags are: • END_CALL_REASON • BALANCE_NAMES • EXCEEDED_BALANCE_NAMES • FAILED_BALANCE_NAMES

Default:

Notes: Any tag listed in the following section will be suppressed.

Example:

```
suppressedEDRTags = [
 "END_CALL_REASON",
 "BALANCE_NAMES",
 "EXCEEDED_BALANCE_NAMES",
 "FAILED_BALANCE_NAMES"
]
```

Editing the File

Open the configuration file on your system using a standard text editor. Do not use text editors, such as Microsoft Word, that attach control characters. These can be, for example, Microsoft DOS or Windows line termination characters (for example, ^M), which are not visible to the user, at the end of each row. This causes file errors when the application tries to read the configuration file.

Always keep a backup of your file before making any changes to it. This ensures you have a working copy to which you can return.

Loading eserv.config Changes

If you change the configuration file, you must restart the appropriate parts of the service to enable the new options to take effect.

Configuring acs.conf for the SLC

Introduction

CCS runs on the ACS subsystem by providing CCS-specific libraries and plug-ins for slee_acs. The configuration options for slee_acs on the SLC are contained in the **acs.conf** file.

When CCS is installed, it automatically configures entries in **acs.conf** to include the plug-in libraries which run basic functionality. This configuration is required in the acsChassis section for the CCS system to run successfully, though it can be changed by qualified engineers under some circumstances.

The following pages contain a description of the **acs.conf** parameters that are specifically relevant to CCS.

For more information about **acs.conf** and plug-in libraries in general, see *ACS Technical Guide*.

acsChassis

The acsChassis section defines how to handle traffic coming in to slee_acs. It defines the traffic processed by a specified service and service loader plug-in library combination. It also defines how slee_acs processes the traffic to each service.

The available parameters are:

`ChassisPlugin`

Syntax:

Description: Chassis plug-ins provide the ACS Control Plan Editor with an expanded interface to its environment.

The ChassisPlugin lines are required to define which chassis action libraries will be available to slee_acs. The CCS chassis action library (*ccsActions* (on page 180)) must be included here.

Type:

Optionality: Required (must be set to include the required CCS library)

Allowed:

Default:

Notes: The interface between the CPE and the Voucher and Wallet Server is implemented using chassis plug-ins. Other uses include external database operations or network access.

One shared library may implement more than one chassis action.

No further configuration is needed to allow the Chassis to load the plug-ins at startup. However, individual plug-ins may have configuration requirements of their own.

For more information about the slee_acs, see *ACS Technical Guide*.

Example:
```
acsChassis
  ChassisPlugin ccsActions.so
```

MacroNodePluginFile

Syntax: `MacroNodePluginFile` *`libraryname`*

Description: The MacroNodePluginFile lines are required to define which feature node libraries will be available in the control plans used by slee_acs. The CCS feature node library (*ccsMacroNodes* (on page 184)) must be included here.

Type:

Optionality: Required (must be set to include the required CCS library)

Allowed:

Default:

Notes: Some plug-in-based feature nodes distributed with CCS are:

- Attempt Termination with Billing node
- Language Select node
- Voucher Recharge node

Example:
```
MacroNodePluginFile ccsMacroNodes.so
```

ServiceEntry

Syntax: `ServiceEntry(`*`ServiceName`*`,`*`NetworkCPSource`*`,`*`LogicalCPSource`*`,`*`PendingTNSource`*`,`*`ConnectCLISource`*`,`*`RedirectingPartyID`*`,`*`OriginalCalledPartyID`*`,`*`libname`*`)`

Description: The ServiceEntry lines are needed to define which services defined in the **SLEE.cfg** are handled by the CCS service loader library (*ccsSvcLibrary* (on page 196)).

Type:

Optionality: Mandatory (must be set to include the required CCS library).

Allowed: For more information about the structure of this configuration option, see acsChassis ServiceEntry Configuration (SCP) in the *ACS Technical Guide*. For more information about the values which can be used in the service element of this configuration, see *Services* (on page 55) in the *Configuration* chapter in *CCS Technical Guide*.

Default:

Notes: Any service defined in **SLEE.cfg** must have a corresponding `ServiceEntry` line configured in **acs.conf**.

Example:
```
ServiceEntry (CCS,ccsSvcLibrary.so)
```

`srf`

Syntax:	`srf (srfName, UseETC=Y|N, Address=IP|nothing, NOA=0|1|2|3|4 typeOfSrf=NAP|other)`
Description:	The name and number of the Specialized Resource Function (or Intelligent Peripheral) is required for each IP on the network.
Notes:	Parsing should continue until no new IPs can be found in the configuration file. This will eliminate the need for a count to be specified in the configuration file for the number of resources available.
Example:	`srf (nap1,UseETC=N,Address=,NOA=3)`

Services

This table describes the valid values for the *ServiceName* array parameter of the ServiceEntry parameter.

acs.conf String	Description
CCS	Use for CCS voice mobile originating.
CCS_ROAM	Use for CCS voice mobile terminating.
CCS_SM_MO	Use for CCS SMS mobile originating.
CCS_SM_MT	Use for CCS SMS mobile terminating.
REVERSE_CCS_SM_MT	Use for CCS SMS mobile terminating with reverse.
CCS_DATA	Use for CCS DATA.
CCS_BPL	Use exact string for BPL task triggers from the SMS.
CCS_BPL*	Use CCS_BPL prefix for services which should trigger xmlTcIf from a third-party interface.

Note: The CCS Service Loader must trigger one of the following service names before it can extract the XMS, MM, or SMS information from the InitialDP:

- CCS_SM_MO
- CCS_SM_MT
- REVERSE_CCS_SM_MT

Example service entries

Here are some example service entries for CCS services in the `acsChassis` section in **acs.conf**.

```
acsChassis
    ...
    ServiceEntry (CCS,GgNnFf,ILcCaAnN,ccsSvcLibrary.so)
    ServiceEntry (CCS_ROAM,cCoOnN,dDfF,dDfF,E,ccsSvcLibrary.so)
    ServiceEntry (CCS_SM_MO,nN,cC,dD,E,ccsSvcLibrary.so)
    ServiceEntry (CCS_SM_MT,dD,cC,dD,E,ccsSvcLibrary.so)
    ServiceEntry (REVERSE_CCS_SM_MT,cC,dD,dD,E,ccsSvcLibrary.so)
    ServiceEntry (CCS_BPL,ccsSvcLibrary.so)
    ServiceEntry (CCS_BPL*,ccsSvcLibrary.so)
    ...
```

Note: For more information about service entry configuration, see acsChassis ServiceEntry Configuration (SLC) in *ACS Technical Guide*.

acsChassis - optional parameters

The parameters in this portion of the `acsChassis` section are optional and may be added when required. Only one entry per parameter is allowed.

`UnknownNOA`

Syntax:
Description: This value is the NOA to be used, to denormalize an outgoing number.
Type: Integer
Optionality:
Allowed:
Default: 65535
Notes:
Example:

`NormalRule`

Syntax: `(incoming NOA,incoming prefix,outgoing NOA,outgoing #digits to strip,prefix to add)`
Description: Enter a conversion rule for each incoming NOA.
Type: Array
Optionality:
Allowed:
Default:
Notes: Incoming prefix can be 'E' to specify the global rule for a given NOA, which will map anything not matched by a prefix.

Outgoing prefix can be 'E' to specify no digits to add to the digit string.

If a minimum parameter is present and a maximum parameter is not provided then only the minimum check is carried out. If a maximum parameter is provided a minimum parameter must be present.

Example:

- (2,E,5,3,E)
- (2,E,5,3,E,1,9)

The second example includes two optional parameters, which denote a size that a number has to be to trigger a rule. The first parameter is the minimum number of digits, and the second the maximum.

acsChassis - variables

The remaining topics explain the variables described in the `acsChassis` section of the **acs.conf** file.

srf_SLEE

Usage:

`srf (`*`IP_name`*`, UseETC=`*`Y/N`*`, Address=`*`address`*`, NOA=`*`noa`*`, TypeOfSrf=t`*`ype`*`)`

Where:

Parameter	Description
IP_name	The IP name to use as a resource name when specifying announcement entries.
`UseETC`	`Y` or `N`. Use `Y` if an external IP is contacted directly from the SLC. This establishes a temporary connection to that IP.

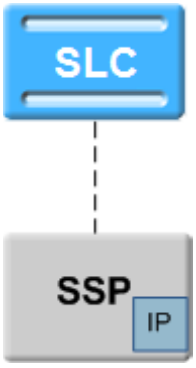
Parameter	Description
Address	Contains the IP address if an external IP is used or nothing if internal
NOA	The nature of address indicator. The indicator is a digit from 0 – 4, as follows: • 0 spare • 1 subscriber number • 2 unknown • 3 national significant number • 4 international significant number
TypeOfSrf	Describes the type of SRF identified by the SRF name. Currently, the only supported value is "NAP". If you do not specify an SRF type then no SRF-type-specific extensions will be activated. **Example:** If you have the UseLanguageExtensions parameter set to Y and you are using a Unisys speaking NAP for announcements, then TypeOfSrf should be NAP, otherwise it should be Other.

Example: `srf (NAP1,UseETC=N,Address=,NOA=3)`

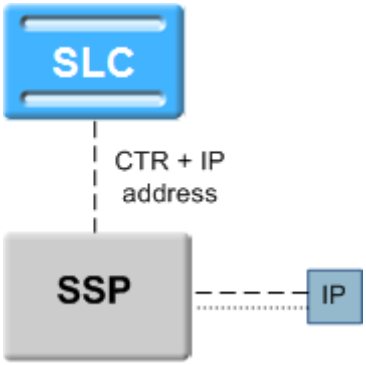
Explanation

There are three ways in which this configuration works, depending on the parameters set:

1 The SLC communicates with the SSP through CTR (Connect to Resource) and using an internal IP.No IP address is required for this option. UseETC is not required (select N). The IP name is required. NOA is required.

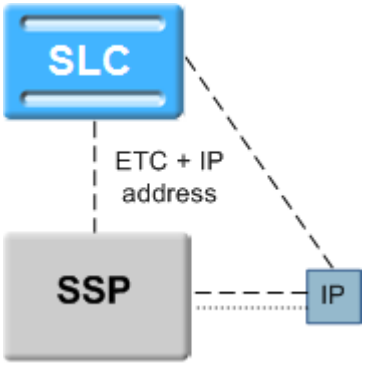


2 The SLC communicates with the SSP through the CTR and IP address. The SSP then uses the IP address to communicate with an external IP. The IP address is required for this option. UseETC is not required (select N). The IP name is required. NOA is required.



3 The SLC communicates with the SSP through the ETC (EstablishTemporaryConnection) and IP address. The SSP then uses the IP address to communicate with an external IP.

The IP address is required for this option. The SLC also communicates directly with the IP, using an ARI (AssistRequestInstructions). UseETC is required (select `Y`). The IP name is required. NOA is required.



NOA and Normal rules

The NOA (nature of address, also known as NOC and NON) is a classification to determine in what realm (local, national or international) a given phone number resides, for the purposes of routing and billing.

Details vary between different implementations of telephone systems, but the following table is representative:

Dialed Digits	NOA	Definition
477 9425	1 ==> subscriber	Number within local telephone exchange
4 477 9425	3 ==> national	Number within country telephone exchange
64 4 477 9425	4 ==> international	Number within world telephone exchange
477 9425	2 ==> UNKNOWN	Numbering scheme rule ==> subscriber
0 4 477 9425	2 ==> UNKNOWN	Numbering scheme rule ==> national
00 64 4 477 9425	2 ==> UNKNOWN	Numbering scheme rule ==> international

In essence, the subscriber's telephone system may try to ascertain the nature by examining the dialed digits. If they can be understood by "built-in" mechanisms, the NOA can unambiguously be one of the values subscriber, national, international, or a finer classification determined by the protocol variant.

Otherwise the NOA is Unknown and the dialed digits must be clarified by a set of (usually simple) rules specified by a numbering scheme.

Leading zeros are used in New Zealand among other places, but the leading characters could be any arbitrary sequence that the numbering scheme could specify.

Ultimately the usage of NOA is determined by the phone network itself which may classify and possibly modify a phone number while it is being transmitted between the service logic and the switch.

People deal with (and database usually store) telephone numbers in their normalized form (for example, 00441918666223). The network gives and receives number in a denormalized form (that is, where the type of number (the Nature of Address) is known explicitly), (for example: [International, 441918666223] from the previous example).

Example:

Normalized number: 049393434

De-Normalized number: Nature of Address: National

Digits: 49393434

Possible Natures of Addresses:

An address can be of the following natures:

Nature of Address	Description
Subscriber (local)	(is 1 with ITU/ETSI CS-1)
Unknown	(is 2 with ITU/ETSI CS-1)
National	(is 3 with ITU/ETSI CS-1)
International	(is 4 with ITU/ETSI CS-1)

Each individual service decides what numbers need to be normalized, however, ACS provides the conversion functionality. The mapping is created through the **acs.conf** file using the following parameters:

Parameter	Description
UnknownNOA *IntegerValue*	This value is the NOA to be used in the code to denormalize a number. The same function is used to normalize as is used to denormalize.
NormalRule *ConversionRule*	This rule determines how to convert between the normal and denormalized number.

The rule is of the following format:

```
incoming NOA ,incoming prefix ,outgoing NOA,outgoing #digits to strip,prefix to add
```

Notes:

- There are NO spaces within the rule.
- Incoming prefix can be 'E' to specify the global rule for a given NOA, which will map anything not matched by a prefix.
- Outgoing prefix can be 'E' to specify no digits to add to the digit string.
- Incoming prefix can be 'E' to specify the global rule for a given NOA, which will map anything not matched by a prefix.
- Outgoing prefix can be 'E' to specify no digits to add to the digit string.

Example 1:

```
UnknownNOA 9999
NormalRule (4,E,9999,0,00)
```

Result:

- Will normalize international Nature Of Address (4) with any prefix(E)
- Will not strip any digits (0), but will prefix 00 to the number
- Value 9999 for the outgoing NOA is ignored as normalized numbers do not have a Nature of Address
- This rule would normalize [International, "6449391234"] to "006449391234".

Example 2:

```
NormalRule (9999,0,3,1,E)
```

Result:

- Will de-normalize (9999 - this must match our UnknownNOA value) numbers beginning with 0.
- Set the Nature of Address to National (3)
- Strip one digit (1) but will not prefix anything (E).
- This rule would de-normalize "049391234" to [National, "49391234"].

Setting up the Screens

About Customizing the UI

You can customize the CCS user interface (UI) by setting Java application properties in the **sms.jnlp** file located in the **/IN/html/** directory. You set JNLP application properties by using the following syntax:

```
<property name="property" value="value" />
```

Where:

- *property* is the name of the application property
- *value* is the value to which that property is set

For more information about the **sms.jnlp** file, see *SMS Technical Guide.*

Java Application Properties

The following application properties are available to customize the UI:

jnlp.ccs.BeORBTimeoutms

Syntax:	`<property name=jnlp.ccs.BeORBTimeoutms value="num" />`
Description:	Specifies the length of time, in milliseconds, after which an ORB request from the screen operator's terminal to the NCC server times out.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	Any positive integer
Default:	20000 (that is, 20 seconds)
Notes:	
Example:	`<property name=jnlp.ccs.BeORBTimeoutms value="5000" />`

jnlp.ccs.defaultEDRSearchAge

Syntax:	`<property name="jnlp.ccs.defaultEDRSearchAge" value="num" />`
Description:	Used to calculate the default start date that is shown in the EDR Viewer. The default start date is equal to the current date and time minus `jnlp.ccs.defaultEDRSearchAge`. The default end date is the current date and time.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	Any positive integer
Default:	2
Notes:	
Example:	`<property name="jnlp.ccs.defaultEDRSearchAge" value="5" />`

jnlp.ccs.defaultEDRSearchCategories

Syntax:	`<property name="jnlp.ccs.defaultEDRSearchCategories" value="list_of_categories" />`
Description:	Specifies the default EDR categories to search for when viewing EDRs in the CCS View EDRs for Subscriber screen. Use a comma-separated string of EDR sub-types.
Type:	String
Optionality:	Optional (default used if not set)

Allowed:

Default: All

Notes: The list of categories must be comma-separated and enclosed in single quotes.

Example:
```
<property name="jnlp.ccs.defaultEDRSearchCategories"
value="'Amount Charge','Bad Pin'" />
```

jnlp.ccs.defaultSubscriberSearchType

Syntax:
```
<property name="jnlp.ccs.defaultSubscriberSearchType"
value="exact|prefix" />
```

Description: Sets the default search type for subscribers in the following locations in the CCS UI:

- The **Subscriber** tab
- The Register Subscriber to Credit Card dialog box

Type: String

Optionality: Optional (default used if not set)

Allowed:

- exact – Searches for the matching subscriber.
- prefix – Searches for all subscribers with IDs that match the entered prefix.

Default: prefix

Notes:

Example:
```
<property name="jnlp.ccs.defaultSubscriberSearchType"
value="exact" />
```

jnlp.acs.ProfileN

Syntax: <property name="jnlp.acs.Profile*number*" value="*new_name*"/>

Description: Specifies to suppress or change the name of any of the 20 profile blocks.

Type: String

Optionality: Optional

Allowed: $1 \leq number \leq 20$

new_name is one of the following:

- – (dash): The profile block is not displayed in screens.
- String comprising any printable characters.

Default: The following table lists default profile block names in the order in which they appear in feature node drop-down lists.

Profile1	VPN Network Profile
Profile2	VPN Station Profile
Profile3	Customer Profile
Profile4	Control Plan Profile
Profile5	Global Profile
Profile6	CLI Subscriber Profile
Profile7	Service Number Profile
Profile8	App Specific 1
Profile9	App Specific 2
Profile10	App Specific 3
Profile11	App Specific 4
Profile12	App Specific 5
Profile13	App Specific 6
Profile14	App Specific 7
Profile15	App Specific 8
Profile16	Any Valid Profile
Profile17	Temporary Storage
Profile18	Call Context
Profile19	Outgoing Extensions
Profile20	Incoming Extensions

Notes:

- If VPN is not installed, Profile1 and Profile2 are suppressed by default.
- If Charging Control Services is installed, profile block names associated with Profile8 through Profile15 are changed automatically. For more information, see *CCS Technical Guide*.
- If RCA is not installed, Profile19 and Profile20 are suppressed by default. You can make them available by installing RCA or by appending them to the **sms.jnlp** file.
- Feature nodes with writable fields cannot write into Profile16.

Examples:

```
<property name="Profile1" value="-" />
<property name="Profile6" value="Originating CLI" />
```

jnlp.ccs.MaxProductTypePeriodicCharges

Syntax:
```
<property name="jnlp.ccs.MaxProductTypePeriodicCharges"
value="int"/>
```

Description: Specifies the maximum number of periodic charges that may be assigned to a product type.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 15

Notes:

Example:
```
<property name="jnlp.ccs.MaxProductTypePeriodicCharges"
value="15"/>
```

jnlp.ccs.ShowEmptyEDRTags

Syntax: `<property name="jnlp.ccs.ShowEmptyEDRTags" value= "taglist" />`

Description: Lists the CCS EDR tags that must be displayed in EDR Viewer or CCP Dashboard when they are empty.

Type: String

Optionality: Optional (default used if not set)

Allowed: Comma separated list of the tags to include.

Default: Empty tags are not displayed in EDR Viewer.

Notes: Do not insert spaces in the list of tags.

Example:
```
<property name="jnlp.ccs.ShowEmptyEDRTags"
value="ACS_CUST_ID,PI,WALLET_TYPE" />
```

jnlp.ccs.showSecondaryBE

Syntax: `<property name="jnlp.ccs.showSecondaryBE" value="value" />`

Description: The number of seconds .....

Type: Integer, Decimal, Array, Parameter list, String, Boolean

Optionality: Optional (default used if not set)

Allowed:

Default:

Notes:

Example: `<property name="jnlp.ccs.showSecondaryBE" value="value" />`

jnlp.ccs.voucherManagement

Syntax: `<property name="jnlp.ccs.voucherManagement" value="?" />`

Description: The number of seconds .....

Type: Integer, Decimal, Array, Parameter list, String, Boolean

Optionality: Optional (default used if not set)

Allowed:

Default:

Notes:

Example: `<property name="jnlp.ccs.voucherManagement" value="?" />`

jnlp.ccs.VRRedeemMaxVoucherLength

Syntax:
```
<property name="jnlp.ccs.VRRedeemMaxVoucherLength"
value="int" />
```

Description: Specifies the maximum number of digits in a voucher number.

Type: Integer

Optionality: Optional (default used if not set)

Allowed: Must be equal to or larger than VRRedeemMinVoucherLength.

Default: 18

Example:
```
<property name="jnlp.ccs.VRRedeemMaxVoucherLength"
value="18" />
```

`jnlp.ccs.VRRedeemMinVoucherLength`

Syntax:	`<property name="jnlp.ccs.VRRedeemMinVoucherlength" value="`*int*`" />`
Description:	Specifies the minimum number of digits in a voucher number.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	Must be equal to or smaller than VRRedeemMaxVoucherLength.
Default:	10
Example:	`<property name="jnlp.ccs.VRRedeemMinVoucherlength" value="10" />`

Defining the Screen Language

Introduction

The default language file sets the language that the Java administration screens start in. The user can change to another language after logging in.

The default language can be changed by the system administrator.

By default, the language is set to English. If English is your preferred language, you can skip this step and proceed to the next configuration task, *Defining the Help Screen Language* (on page 65).

Default.lang

When CCS is installed, a file called **Default.lang** is created in the application's language directory in the screens module. This contains a soft-link to the language file which defines the language which will be used by the screens.

If a **Default.lang** file is not present, the **English.lang** file will be used.

The CCS **Default.lang** file is **/IN/html/Ccs_Service/language/Default.lang**

Example Screen Language

If Dutch is the language you want to set as the default, create a soft-link from the **Default.lang** file to the **Dutch.lang** file.

Procedure

Follow these steps to set the default language for your CCS Java Administration screens.

Step	Action
1	Change to the following directory: **/IN/html/Ccs_Service/language** **Example command:** `cd /IN/html/Ccs_Service/language`
2	Ensure the Default.lang file exists in this directory.
3	If the required file does not exist, create an empty file called Default.lang.

Step	Action

4	Ensure that the language file for your language exists in this directory. The file should be in the format:
	`language.lang`
	Where:
	language = your language.
	Example:
	`Spanish.lang`

5	If the required language file does not exist, either: • create a new one with your language preferences, or • contact Oracle support. To create a language file, you will need a list of the phrases and words used in the screens. These should appear in a list with the translated phrase in the following format: `original phrase=translated phrase` Any existing language file should have the full set of phrases. If you do not have an existing file to work from, contact Oracle support with details.
6	Create a soft link between the Default.lang file, and the language file you want to use as the default language for the CCS Java Administration screens. **Example command:** `ln -s Dutch.lang Default.lang`

Defining the Help Screen Language

Introduction

The default Helpset file sets the language that the help system for the Java Administration screens start in. The user can change to another language after logging in.

The default language can be changed by the system administrator. By default, the language is set to English.

Default_Ccs_Service.hs

When CCS is installed, a file called **Default_Ccs_Service.hs** is created in the application's language directory in the screens module. This contains a soft-link to the language file which defines the language which will be used by the screens.

If a **Default_Ccs_Service.hs** file is:

- Not present, the **English_Ccs_Service.hs** file will be used.
- Present, the default language will be used.

The **Default_Ccs_Service.hs** file is **/IN/html/Acs_Service/helptext/Default_Ccs_Service.hs**.

Example helpset language

If Dutch is the language you want to set as the default, create a soft-link from the **Default_Ccs_Service.hs** file to the **Dutch_Ccs_Service.hs** file.

Setting the default language for your CCS graphical user interface

Follow these steps to set the default language for your CCS graphical user interface.

Step	Action
1	Change to the following directory: **/IN/html/Ccs_Service/helptext** **Example command:** `cd /IN/html/Ccs_Service/helptext`
2	Ensure the **Default_Ccs_Service.hs** file exists in this directory.
3	If the required file does not exist, create an empty file called **Default_Ccs_Service.hs**.
4	Ensure that the language file for your language exists in this directory. The file should be in the format: *language*`_Ccs_Service.hs` **Where:** *language* = your language. **Example:** **Dutch_Ccs_Service.hs**
5	If the required language file does not exist, perform one of the following actions: • Create a new one with your language preferences • Contact Oracle support To create a language file, you will need a list of the phrases and words used in the UI. These should appear in a list with the translated phrase in the following format: *original phrase=translated phrase* Any existing language file should have the full set of phrases. If you do not have an existing file to work from, contact Oracle support with details.
6	Create a soft link between the **Default_Ccs_Service.hs** file, and the language file you want to use as the default language for the SMS UI. **Example command:** `ln -s Dutch_Ccs_Service.hs Default_Ccs_Service.hs`

Configuration Through the ACS Screens

Introduction

Some CCS functions rely on resources which are configured through the ACS UI.

ACS resources

This table lists the resources which may need to be configured through the ACS UI in order to be able to configure CCS.

Resource	ACS Screen
ACS customers, including resource limits.	ACS Customer
Sets, including geography, holiday, announcement, VARS, VARS mapping and feature sets.	ACS Configuration
Notification templates.	ACS Configuration
Control plans	Control Plan Editor

Adding announcement sets automatically

NCC can provide a customized SQL script that adds an entire announcement set.

This script is run once at installation, from SMS as sms_oper.

If you wish to use this script then contact your Oracle account manager.

User Interface-Based Configuration Tasks

Introduction

Some of the configuration for CCS must be completed through the SMS, ACS and CCS UI windows.

For more information about using the CCS UI, see *Charging Control Services User's Guide*.

SMS UI configuration

This table lists elements of the system which you may need to configure through the SMS UI.

Element	Description of Configuration
Replication	Ensure CCS tables will be correctly replicated to the appropriate nodes in the IN.
Users	Setting up different levels of access for system administrators.
Alarms	Setting up filtering and monitoring systems for CCS alarms.
Statistics	Setting up statistics which relate to the nodes which CCS runs on.

For more information about using the SMS UI, see *SMS User's Guide*.

ACS UI configuration

This table lists elements that you may need to configure through the ACS UI.

Element	Description of Configuration
ACS customers	All calls are processed in relation to an ACS customer. ACS customers are used to manage control plans and resources.
Resource sets	Resource sets are required for much of the functionality used in control plans. In particular, resource sets define: • Geographic regions • Holidays • Announcements • Feature node sets
Control plans	Call processing logic is defined in control plans.
Statistics	Setting up statistics for the control plans used in CCS.

For more information about:

- Using the ACS UI, see *Advanced Control Services User's Guide.*
- The Control Plan Editor, see *CPE User's Guide*.
- The available feature nodes, see *Feature Nodes Reference Guide*.

CCS UI configuration

This table lists elements of the system which you may need to configure through the CCS UI.

Element	Description of Configuration
Currencies	Currencies must be set up for financial processes.

For more information about using the CCS UI, see *Charging Control Services User's Guide*.

Configuring VWS processes for CCS

VWS processes used by CCS

There are a number of VWS processes which must be configured correctly for CCS to use the VWS functionality:

- BeClient interface on the SLC must be configured to include CCS plug-ins
- beVWARS on the VWS must be configured to include the CCS beVWARS plug-ins and message handlers
- beServer VWS must be configured to include the CCS beServer plug-ins

For more information about configuring these processes, see:

- *Background Processes on the SLC* (on page 173)
- *Background Processes on the VWS* (on page 209)

Message handlers and event plug-ins

Message handlers provide functionality which is specifically related to messages passed between BeClient and the VWS. Plug-ins are designed to handle specific events such as a balance expiry date being passed.

Message handlers

CCS installs a number of message handlers and plug-ins into the VWS for handling the CCS-specific messages and functionality. This table lists the main message handlers installed for *beVWARS* (on page 210).

Message Handler	Description
ccsVWARSWalletHandler (on page 274)	This beVWARS plug-in handles inquiries/updates to wallets and balances.
ccsVWARSReservationHandler (on page 263)	This beVWARS plug-in handles call-related messages.
ccsVWARSNamedEventHandler (on page 252)	This beVWARS plug-in handles named event-related messages.

These handlers, and their respective configuration items, are described in *Background Processes on the VWS* (on page 209).

The ccsVWARSVoucherHandler is described in *Voucher Manager Technical Guide*.

BeClient IF

The BeClient is covered in more detail in *VWS Technical Guide*. However it needs to be configured for CCS to allow functions such as wallet interaction.

For more information about configuring BeClient for CCS, see *BeClient* (on page 173).

Configuring CCS Macro Nodes

Introduction

Macro nodes are feature nodes that are used by CCS using the ACS Control Plan Editor. Macro nodes are supplied by many Oracle applications and require the presence of ACS for use.

Macro nodes require some configuration to be entered into the **eserv.config** file. The following sections will detail the configuration that is necessary for the CCS macro nodes.

The macro node reads the global configuration file (**eserv.config**) on initialization. Should the configuration of a macro node be changed, the configuration files must be re-read.

Macro Node location

Macro nodes are delivered as shared libraries, and are located on installation in:

/IN/service_packages/CCS/lib/

Node icons are installed in:

/IN/html/Acs_Service/images/

Macro Node icons

Node icons are delivered as gif files and are named according to the following standard:

Filename	Description
FN*macroNodeNamfor* exampleif	The icon that appears on the node in the CPE.
LFN*macroNodeName.gif*	The icon that appears in the edit dialog for the specific feature node.
PFN*macroNodeName*.gif	The icon that appears in the CPE feature node palette.

eserv.config Macro Node configuration

This is a high level view of the `ccsMacroNodes` configuration section of **eserv.config**.

```
CCS = {
    ccsMacroNodes = {
        general macro node config
        macro node config for specific node
        MacroNodeName = {
            configuration for specific macro node
        }
    }
}
```

See *ccsMacroNodes* (on page 184) for specific macro node configuration.

Introduction

To calculate the caller's wallet balance a configurable list of balance types will be checked. The list of balance types to be checked for each customer is configured in the SLC's **eserv.config** file. If the list of balance types for the balance status feature node is omitted from the **eserv.config**, only the default balance type will be checked. If included, the default balance type will only be checked if it appears in the list.

A section like the one below must be placed in the CCS section of the file:

```
CCS = {
    ccsMacroNodes = {
        BSBCheckBalanceTypes = [
            { acsCustomerId = customer_id_1
                balTypeIds = [
                    balancetype_id_1, balancetype_id_2, balancetype_id_3
                ]
            }
            { acsCustomerId = customer_id_2
                balTypeIds = [
                    balancetype_id_4, balancetype_id_5
                ]
            }
        ]
    }
}
```

acsCustomerId

Syntax: See the Balance Status Branch Introduction.

Description: This is the ID of the ACS customer in the database.

Type:

Optionality:

Allowed: The acsCustomerId must exist in the ACS_CUSTOMER database table.

Default: 1

Notes:

Example:

balTypeIds

Syntax: See the Balance Status Branch Introduction.

Description: The database ids of the balance types that are to be checked for each customer.

Type:

Optionality:

Allowed:

Default: None

Notes: The balTypeIds listed must exist in the CCS_BALANCE_TYPE database table.

Example:

BSBCheckBalanceTypes

Syntax: See the Balance Status Branch Introduction.

Description: The specific balance types that are to be checked for each customer.

Type: Array

Optionality: Optional. If there is no `BSBCheckBalanceTypes` section for the current customer then only the default balance type is used to determine if the caller has credit. If there is a `BSBCheckBalanceTypes` section for the current customer then the total of all of the balance types specified is used to determine if the caller has credit.

Allowed:

Default: None

Notes: The balance types must all have the same balance unit.

Example:

Switch Configuration for the UATB Node

Switch configuration

The switch types used to control the switch communication flows for the UATB feature node are defined in the `acsCharging.switchConfiguration` section of the **eserv.config** configuration file.

acsCharging.switchConfiguration

Several switch types may be defined and the chassis action GetSwitchParameters determines which switch is in use for a particular call.

Example:

```
acsCharging = {
   switchConfiguration = [
       {
           switchType = "cap3"
           addDisconnectOrRelease = false
       }
       {
           # INTERNAL switch type
           # default IDP appContext {1,3,6,1,4,1,3512,10,100}

           switchType = "internal"
           addDisconnectOrRelease = true
       }

       {
           switchType = "internal"
           addDisconnectOrRelease = true

           extended = {
              # extended IDP appContext {1,3,6,1,4,1,3512,10,100,2}
              oid = 2
              allowZeroSecondsApplyCharge = true
           }
       }
  ]
```

The available parameters are:

`addContinue`

Syntax:	`addContinue = true|false`
Description:	Defines whether the UATB feature node should enable send responses, add responses, and continue responses to the TCAP to enable charging for a successful subsequent reservation on the Voucher and Wallet Server.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true, false
Default:	false
Example:	`addContinue = false`

`addDisconnectOrRelease`

Syntax:	`addDisconnectOrRelease = true|false`
Description:	Defines whether the UATB node can release or disconnect calls during billing scenarios. For example, where the call is still active but the calling party has exhausted their funds or the maximum call limit has been reached.

Type: Boolean
Optionality:
Allowed: true, false
Default: false
Notes:
Example: `addDisconnectOrRelease = false`

switchType

Syntax: `switchType = "type"`
Description: Specifies a switch type for a UATB node.
Type: String
Optionality: Optional
Allowed: One of:
- cap2
- cap3
- internal
- nokia

Default: Not set
Notes: Use the internal switch type to support the extra information passed by the Diameter Control Agent (DCA) to ACS in the IDP extension fields in Continue and Release Call operations.
Example: `switchType = "internal"`

allowZeroSecondsApplyCharge

Syntax: `allowZeroSecondsApplyCharge = value`
Description: The chassis that the switch can handle time grants of zero deciseconds.
Type: Integer, Decimal, Array, Parameter list, String, Boolean
Optionality: Optional (default used if not set)
Allowed: true, false
Default: true
Notes:
Example: `allowZeroSecondsApplyCharge = true`

oid

Syntax: `oid = value`
Description: The extension digit number {1,3,6,1,4,1,3512,10,100,2}.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default:
Notes:
Example: `oid = 2`

Voucher Status Report Configuration

Introduction

voucherStatusReport.env provides configuration for the Voucher Status report in addition to the configuration available at *VoucherStatus* (on page 165).

For more information about the Voucher Status report, see *Charging Control Services User's Guide*.

Parameters

The following parameters can be used in **voucherStatusReport.env**.

TZ_CODE

Syntax:	TZ_CODE="*TZ*" export TZ_CODE
Description:	The timezone to use when calculating the dates to print in the report.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	Any valid Unix timezone code.
Default:	GMT
Notes:	Used for converting date in GMT to an appropriate timezone.
Example:	TZ_CODE="GMT" export TZ_CODE

VR_MSISDN_LENGTH

Syntax:	VR_MSISDN_LENGTH=*int* export VR_MSISDN_LENGTH
Description:	The maximum number of characters in an MSISDN printed in the report.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	20
Notes:	Any MSISDN longer than this number will have the final digits removed.
Example:	VR_MSISDN_LENGTH=20 export VR_MSISDN_LENGTH

VR_STATUS

Syntax:	VR_STATUS="*NORMAL*|*SPECIAL*" export VR_STATUS
Description:	How the voucher status should be presented in the report.
Type:	String
Optionality:	Optional (default used if not set).

Allowed:	NORMAL	Use the normal status letters: • R - Redeemed • A - Active • F - Frozen • C - Created
	SPECIAL	Use alternative status letters: • R -> A - Acredita • A -> D - Disponsible • F -> B - Bloqueado • C -> G - Generada
Default:	NORMAL	
Notes:		
Example:	`VR_STATUS="SPECIAL"` `export VR_STATUS`	

Example

This text shows an example of the **voucherStatusReport.env** configuration file.

```
#!/bin/sh

VR_MSISDN_LENGTH=20
export VR_MSISDN_LENGTH

VR_STATUS="NORMAL"
export VR_STATUS

TZ_CODE="GMT"
export TZ_CODE
```

CCP Configuration

Introduction

The Customer Care Portal (CCP) is a WebStart application that provides a customized view of CCS subscribers.

Note: For more information about the CCP, see *Customer Care Portal User's Guide*.

ccp.jnlp File

The **ccp.jnlp** file is used to start the CCP. It contains the following properties that can be configured for a specific customer:

- The customer logo displayed in the CCP Login screen
- Whether to cache user names and passwords or to force users to login fresh each time
- If caching is allowed, the port on which to start a listening service
- The service provider initially displayed in the **Service Provider** selection box in the CCP Dashboard screen
- The maximum number of entries on the History panel of the CCP Dashboard

Application properties use the following format:

```
<property name= "property" value="value"/>
```

Where:

- *property* is the name of the Java application property
- *value* is the value of the Java application property

`jnlp.ccs.AllowDeletedVouchers`

Syntax: `<property name="jnlp.ccs.allowDeletedVouchers" value="value" />`

Description: Specifies whether you can set a voucher status or a voucher range state to Deleted.

This parameter is used by the following in the Voucher Manager screens:

- The **Vouchers** tab
- The **Voucher Ranges** tab

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

- True
- t(rue)
- Yes
- y(es)
- 1

All other values are considered to be false.

Default: True

Notes: If set to:

- True – You can set a voucher range state or a voucher status to Deleted.
- False – You cannot set a voucher range state or a voucher status to Deleted.

Example:
```
<property name="jnlp.ccs.allowDeletedVouchers" value="true" />
```

`ccp.CustomerLogo`

Syntax: `<property name="ccp.CustomerLogo" value = "filename" />`

Description: Use to display a different graphic in the CCP login screen to the one installed with the system.

Type: String

Optionality: Optional (default used if not set).

Allowed: gif or jpeg files.

Default: ccp/oracle.gif

Notes: If the specified file does not exist, then the default is used.

Example:
```
<property name="ccp.CustomerLogo" value = "ccp/oracle.gif"
/>
```

`jnlp.ccp.dashboardPort`

Syntax: `<property name="jnlp.ccp.dashboardPort" value="address" />`

Description: When caching is allowed, specifies the port on which to start a listening service.

Type: String

Optionality: Required when `jnlp.ccp.ForceLogin` is true.

Allowed:

Default: 7007

Notes:

Example:
```
<property name="jnlp.ccp.dashboardPort" value="1234" />
```

jnlp.sms.dbPassword

Syntax: `<property name="jnlp.sms.dbPassword" value="password" />`

Description: Specifies the database password. This password is for a special database user that the ACS Logon screen uses before the user logs in. This property is set during installation and is then not changed.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: acs_public

Notes: Do not change this value.

Example: `<property name="jnlp.sms.dbPassword" value="acs_public" />`

jnlp.sms.dBUser

Syntax: `<property name="jnlp.sms.dBUser" value="user" />`

Description: Specifies the database user name. This is a special database user that the ACS Logon screen uses before the user logs in. This property is set during installation and is then not changed.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: acs_public

Notes: Do not change this value.

Example: `<property name="jnlp.sms.dBUser" value="acs_public" />`

jnlp.ccs.defaultEDRSearchAge

Syntax: `<property name="jnlp.ccs.defaultEDRSearchAge" value="num" />`

Description: Used to calculate the default start date that is shown in the EDR Viewer. The default start date is equal to the current date and time minus `jnlp.ccs.defaultEDRSearchAge`.

The default end date is the current date and time.

Type: String

Optionality: Optional (default used if not set)

Allowed: Any positive integer

Default: 2

Notes:

Example: `<property name="jnlp.ccs.defaultEDRSearchAge" value="5" />`

jnlp.ccs.defaultEDRSearchCategories

Syntax: `<property name="jnlp.ccs.defaultEDRSearchCategories" value="list_of_categories" />`

Description: Specifies the default EDR categories to search for when viewing EDRs in the CCS View EDRs for Subscriber screen.

Use a comma-separated string of EDR sub-types.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: All

Notes: The list of categories must be comma-separated and enclosed in single quotes.

Example:
```
<property name="jnlp.ccs.defaultEDRSearchCategories"
value="'Amount Charge','Bad Pin'" />
```

jnlp.ccp.ForceLogin

Syntax:
```
<property name="jnlp.ccp.ForceLogin" value="Y|N" />
```

Description: Specifies whether to allow caching of user names and passwords or to force users to login fresh each time.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:
- Y – The user must log on each time to start a new session.
- N – A small window running on the user's machine starts the screens by using the `jnlp.ccp.dashboardPort` resource to request each new session.

Default: N

Notes:

Example:
```
<property name="jnlp.ccp.ForceLogin" value="N" />
```

jnlp.sms.host

Syntax:
```
<property name="jnlp.sms.host" value="IPaddress" />
```

Description: Specifies the Internet Protocol (IP) address for the SMS host machine that is set at installation.

Type: String

Optionality: Required

Allowed:
- IP version 4 (IPv4) addresses
- IP version 6 (IPv6) addresses

Default: No default

Notes: You can use the industry standard for omitting zeros when specifying IP addresses.

Examples:
```
<property name="jnlp.sms.host" value="192.0.2.0" />
<property name="jnlp.sms.host"
value="2001:db8:0000:1050:0005:0600:300c:326b" />
<property name="jnlp.sms.host"
value="2001:db8:0:0:0:500:300a:326f" />
<property name="jnlp.sms.host" value="2001:db8::c3" />
```

jnlp.ccp.maxHistory

Syntax:
```
<property name="jnlp.ccp.maxHistory" value="number" />
```

Description: Sets the maximum number of items that can be listed on the History panel in the CCP Dashboard.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 20

Notes:

Example: `<property name="jnlp.ccp.maxHistory" value = "20" />`

jnlp.sms.namingServerPort

Syntax: `<property name="jnlp.sms.namingServerPort" value="port" />`
Description: Tells the Dashboard screens how to contact the naming server.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: 5556
Notes: The value in this field should be the same as the value set in the -p parameter in **/IN/service_packages/SMS/bin/smsNamingServerStartup.sh**.
Example: `<property name="jnlp.sms.namingServerPort" value="5556" />`

jnlp.ccp.normaliseFile

Syntax: `<property name="jnlp.ccp.normaliseFile" value = "filename" />`
Description: Specifies the location and name of the file that contains the set of CCP normalization rules.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: ccp/normalise.config
Notes:
Example:
```
<property name="jnlp.ccp.normaliseFile"
value="ccp/normalise.config" />
```

jnlp.ORB_HOST

Syntax: `<property name="jnlp.ORB_HOST" value="hostsms" />`
Description: Specifies the host name of the machine running the ccsBeOrb background process.
Type: String
Optionality: Optional (default used if not set)
Allowed:
Default: The SMS machine host name.
Notes:
Example: `<property name="jnlp.ORB_HOST" value="hostname" />`

jnlp.sms.port

Syntax: `<property name="jnlp.sms.port" value="num" />`
Description: Specifies the SQL*Net port for connecting to the SMS host machine.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 1521
Notes: Set at installation
Example: `<property name="jnlp.sms.port" value="1521" />`

ccp.ServiceProvider

Syntax: `<property name="ccp.ServiceProvider" value = "name" />`

Description: The initial service provider to display in the **Service Provider** selection box in the CCP Dashboard screen.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: Boss

Notes:

Example: `<property name="ccp.ServiceProvider" value = "Boss" />`

jnlp.sms.sslCipherSuites

Syntax:
```
<property name = "jnlp.sms.sslCipherSuites"
value="(TLS_RSA_WITH_AES_128_CBC_SHA)" />
```

Description: Specifies the cipher suites to use for SSL encryption. You must set this property if you are using encrypted SSL for connecting to the SMS database.

Type: String

Optionality: Optional (default used if not set)

Allowed: (TLS_RSA_WITH_AES_128_CBC_SHA)

Default: (TLS_RSA_WITH_AES_128_CBC_SHA)

Notes: You must also set the SSL_CIPHER_SUITES property to (TLS_RSA_WITH_AES_128_CBC_SHA) in the **listener.ora** and **sqlnet.ora** files.

Example:
```
<property name = "jnlp.sms.sslCipherSuites"
value="(TLS_RSA_WITH_AES_128_CBC_SHA)" />
```

jnlp.trace

Syntax: `<property name="jnlp.trace" value="value" />`

Description: Specifies whether to enable tracing for the Control Plan Editor. The output is displayed in the Java Console.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: on | off, true | false, yes | no, 1 | 0, enabled | disabled

Default: Off

Notes:

Example: `<property name="jnlp.trace" value="on" />`

jnlp.sms.TZ

Syntax: `<property name="jnlp.sms.TZ" value="timezone" />`

Description: Specifies the time zone used for all time and date values displayed in NCC UI windows.

Type: String

Optionality: Optional (default used if not set)

Allowed: Any Java supported time zone.

Default: GMT

Notes: For a full list of Java supported time zones, see Time Zones.

Example: `<property name="jnlp.sms.TZ" value="GMT" />`

Example ccp.jnlp Resource Properties

The following example configuration shows CCP resources in the **ccp.jnlp** file.

Note: The **ccp.jnlp** file is located in the **/IN/html/ccp/cgi-bin/** directory on the SMS.

```
<resources>
   <j2se version="1.8.0+" href="http://java.sun.com/products/autodl/j2se"  />
   <property name="jnlp.packEnabled" value="true" />
   <jar href="ccs.sig.jar" main="true" />
   <jar href="ojdbc6.sig.jar" />
   <jar href="acs.sig.jar" />
   <jar href="sms.sig.jar" />
   <jar href="common.sig.jar" />
   <property name="ccp.ServiceProvider" value="Boss" />
   <property name="jnlp.sms.namingServerPort" value="5556" />
   <property name="ccp.CustomerLogo" value="SMS/images/oracleNCC.png" />
   <property name="jnlp.ccp.maxHistory" value="20" />
   <property name="ccp.normaliseFile" value="ccp/normalise.config" />
   <property name="jnlp.sms.host" value="IPADDR" />
   <property name="jnlp.sms.databaseID" value="port:SMF" />
   <property name="jnlp.sms.TZ" value="GMT" />
   <property name="dashboardPort" value="7007" />
   <property name="jnlp.ccp.ForceLogin" value="N" />
   <extension name="Java Help" href="ohj.jnlp" />
</resources>
```

The following application properties, defined in the **ccp.jnlp** file, are defined in the **sms.jnlp** file. You must set the application properties in the **ccp.jnlp** file and the **sms.jnlp** file to the same value.

Note: For more information about the **sms.jnlp** application properties, see *SMS Technical Guide*.

```
<resources>
   <property name="jnlp.ORB_HOST" value="hostsmp" />
   <property name="jnlp.sms.host" value="192.168.26.22" />
   <property name="jnlp.sms.databaseID" value="1533:SMF" />
   <property name="jnlp.sms.TZ" value="GMT" />
   <property name="jnlp.ccs.defaultEDRSearchAge" value="20"/>
   <property name="jnlp.ccs.defaultEDRSearchCategories" value="'Amount Charge','Bad Pin'" />
<resources/>
```

CCP Application Properties for SSL and Non-SSL Database Connections

The following Java application properties in the **ccp.jnlp** file are used for SSL and non-SSL connections to the database:

`jnlp.sms.database`

Syntax:	`<property name="jnlp.sms.database" value="SMF" />`
Description:	Specifies the Oracle SID for the SMF database.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	SMF
Notes:	Set at installation.
Example:	`<property name="jnlp.sms.database" value="SMF" />`

jnlp.sms.databaseHost

Syntax: `<property name="jnlp.sms.databaseHost" value = "ip:port:sid" />`

Description: Sets the IP address and port to use for non-SSL connections to the SMF database, and the database SID.

- To use non-SSL connections to the database, set *port* to 1524 and the `jnlp.sms.EncryptedSSLConnection` property to false.
- To use SSL connections to the database, set the `jnlp.sms.EncryptedSSLConnection` property to true and set either the `jnlp.sms.secureConnectionDatabaseHost` property or the `jnlp.sms.secureConnectionClusterDatatbaseHost` property appropriately. When the `jnlp.sms.EncryptedSSLConnection` property is set to true or is undefined, `jnlp.sms.databaseHost` is ignored.

Type: String

Optionality: Optional

Allowed:

Default: Not set. Secure SSL connection is enabled at installation by default.

Notes: Internet Protocol version 6 (IPv6) addresses must be enclosed in square brackets []; for example: **`[2001:db8:`***`n:n:n:n:n:n`***`]`** where *n* is a group of 4 hexadecimal digits. The industry standard for omitting zeros is also allowed when specifying IP addresses.

Examples:

```
<property name="jnlp.sms.databaseHost" value =
"192.0.2.1:2484:SMF" />
<property name="jnlp.sms.databaseHost" value =
"[2001:db8:0000:1050:0005:0600:300c:326b]:2484:SMF" />
<property name="jnlp.sms.databaseHost" value =
"[2001:db8:0:0:0:500:300a:326f]:2484:SMF" />
<property name="jnlp.sms.databaseHost" value =
"[2001:db8::c3]:2484:SMF" />
```

jnlp.sms.databaseID

Syntax: `<property name="jnlp.sms.databaseID" value="port:sid" />`

Description: Specifies the SQL*Net port for connecting to the database, and the database SID.

Type: String

Optionality: Required

Allowed:

Default: 1521:SMF

Notes:

- To use non-SSL connections to the database, set *port* to 1521 and the `jnlp.sms.EncryptedSSLConnection` property to false.
- To use SSL connections to the database, set the `jnlp.sms.EncryptedSSLConnection` property to true and set either the `jnlp.sms.secureConnectionDatabaseHost` property or the `jnlp.sms.secureConnectionClusterDatatbaseHost` property appropriately. When the `jnlp.sms.EncryptedSSLConnection` property is set to true or is undefined, `jnlp.sms.databaseID` is ignored.

Example:

```
<property name="jnlp.sms.databaseID" value="1521:SMF" />
```

jnlp.sms.clusterDatabaseHost

Syntax:
```
<property name="jnlp.sms.clusterDatabaseHost" value =
"(DESCRIPTION=
(LOAD_BALANCE=YES)(FAILOVER=ON)(ENABLE=BROKEN)
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=type)(HOST=name)(PORT=port))
(ADDRESS=(PROTOCOL=type)(HOST=name)(PORT=port)))
(CONNECT_DATA=(SERVICE_NAME=SMF)(FAILOVER_MODE=(TYPE=SESSION)
(METHOD=BASIC)(RETRIES=5)(DELAY=3))))" />
```

Description: Specifies the connection string (including a host and an alternative host address, in case the first IP address is unavailable) for non-SSL cluster-aware connection to the database.

To use non-SSL connections to the database, set the `jnlp.sms.EncryptedSSLConnection` property to false.

Type: String

Optionality: Optional

Allowed:

Default: By default, *port* is set to 1521.

Notes: If present, this property is used instead of the `jnlp.sms.databaseID` property.

Example:
```
<property name="jnlp.sms.clusterDatabaseHost" value =
"(DESCRIPTION=
(LOAD_BALANCE=YES)(FAILOVER=ON)(ENABLE=BROKEN)
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=smsphysnode1)
(PORT=1521))
(ADDRESS=(PROTOCOL=TCP)(HOST=smsphysnode2)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME=SMF)(FAILOVER_MODE=(TYPE=SESSION)
(METHOD=BASIC)(RETRIES=5)(DELAY=3))))" />
```

jnlp.sms.EncryptedSSLConnection

Syntax:
```
<property name="jnlp.sms.EncryptedSSLConnection" value = "value"
/>
```

Description: Specifies whether connections to the client UI use encrypted SSL.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true – Use encrypted SSL connections to access the client UI.

false – Use non-SSL connections to access the client UI.

Default: true

Notes:

- To use SSL connections to the database, set the `jnlp.sms.EncryptedSSLConnection` property to true and set either the `jnlp.sms.secureConnectionDatabaseHost` property or the `jnlp.sms.secureConnectionClusterDatatbaseHost` property appropriately.
- To use non-SSL connections to the database, set the `jnlp.sms.EncryptedSSLConnection` property to false.

Example:
```
<property name="jnlp.sms.EncryptedSSLConnection" value = "true"
/>
```

jnlp.sms.sslCipherSuites

Syntax: `<property name = "jnlp.sms.sslCipherSuites" value="(TLS_RSA_WITH_AES_128_CBC_SHA)" />`

Description: Specifies the cipher suites to use for SSL encryption. You must set this property if you are using encrypted SSL for connecting to the SMS database.

Type: String

Optionality: Optional (default used if not set)

Allowed: (TLS_RSA_WITH_AES_128_CBC_SHA)

Default: (TLS_RSA_WITH_AES_128_CBC_SHA)

Notes: You must also set the SSL_CIPHER_SUITES property to (TLS_RSA_WITH_AES_128_CBC_SHA) in the **listener.ora** and **sqlnet.ora** files.

Example:
```
<property name = "jnlp.sms.sslCipherSuites"
value="(TLS_RSA_WITH_AES_128_CBC_SHA)" />
```

jnlp.sms.secureConnectionDatabaseHost

Syntax:
```
<property name="jnlp.sms.secureConnectionDatabaseHost" value =
"(DESCRIPTION=
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=type)(HOST=IPaddress)
(PORT=port))))(CONNECT_DATA=(SERVICE_NAME=servicename)))" />
```

Description: Specifies the connection string (including host address and port) for encrypted SSL connections to the SMF database on a non-clustered system.

To use SSL connections to the database, set *port* to 2484 and set the `jnlp.sms.EncryptedSSLConnection` property to true.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default:

Notes: If present, this property is used instead of the `jnlp.sms.databaseID` property.

Example:
```
<property name="jnlp.sms.secureConnectionDatabaseHost" value =
"(DESCRIPTION=
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCPS)(HOST=192.0.1.1)
(PORT=2484))))(CONNECT_DATA=(SERVICE_NAME=SMF)))" />
```

jnlp.sms.secureConnectionClusterDatabaseHost

Syntax:
```
<property name="jnlp.sms.secureConnectionClusterDatabaseHost"
value = "(DESCRIPTION=
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=type)(HOST=IPaddress)
(PORT=port))
(ADDRESS=(PROTOCOL=type)(HOST=IPaddress)(PORT=port)))
(CONNECT_DATA=(SERVICE_NAME=servicename)))" />
```

Description: Specifies the connection string (including host address and port) for encrypted SSL connections to the SMF database on a clustered system.

To enable secure SSL connections to the database, set *port* to 2484 and set the `jnlp.sms.EncryptedSSLConnection` property to true.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default:

Notes: If present, this property is used instead of the `jnlp.sms.secureConnectionDatabaseHost` property.

Example:

```
<property name="jnlp.sms.secureConnectionClusterDatabaseHost"
value = "(DESCRIPTION=
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCPS)(HOST=192.0.1.1)
(PORT=2484))
(ADDRESS=(PROTOCOL=TCP)(HOST=192.0.2.1)(PORT=2484)))
(CONNECT_DATA=(SERVICE_NAME=SMF)))" />
```

Setting the Initial Service Provider

Follow these steps to set the initial service provider displayed in the **Service Provider** selection box in the CCP Dashboard screen.

Step	Action
1	Log in to the SMS as the root user.
2	Open the **/IN/html/ccp/cgi-bin/ccp.jnlp** file in a text editor.
3	Enter the name of the initial service provide in the `ccp.ServiceProvider` application property. For example: `<property name="ccp.ServiceProvider" value="Boss" />`
4	Save and close the file.

Customizing the CCP Login Screen

Follow these steps to change the image displayed in the CCP Login screen.

Step	Action
1	Log in to the SMS as the root user.
2	Open the **/IN/html/ccp/cgi-bin/ccp.jnlp** file in a text editor.
3	Add the name of the new image file to the `ccp.CustomerLogo` application property. For example: `<property name="ccp.CustomerLogo" value="ccp/company.gif" />` **Note:** The image can be either a JPEG or a GIF file.
4	Save and close the file

Setting the Maximum History Shown

Follow these steps to set the maximum number of items shown in the History panel of the CCP Dashboard.

Step	Action
1	Log in to the SMS as the root user.
2	Open the **/IN/html/ccp/cgi-bin/ccp.jnlp** file in a text editor.
3	Enter the maximum number of history items to display. For example: `<property name="jnlp.ccp.maxHistory" value="20" />` **Note:** The value specified applies to both subscriber and voucher histories.
4	Save and close the file.

normalise.config Configuration File

The **normalise.config** file contains the set of normalization rules for prefixes used in the CCP Dashboard. The file is located in the **IN/html/ccp** directory on the SMS.

Normalization rules in the file use the following format:

```
PREFIX NUM-STRIP,DIGITS-ADD MIN-LENGTH,MAX-LENGTH
```

Here is an example **normalise.config** file:

```
44 2,0
00 2,01
000 3,21
21 2,00
```

For example, rule "44 2,0" specifies to replace the prefix '44' with '0'.

Apache Configuration

As part of the "login once" for accessing the dashboard, the APACHE server requires additional configuration (see *SMS Technical Guide* for more information about Apache server installation and configuration).

Follow these steps to configure the Apache daemon for the dashboard:

Step	Action
1	Open the **httpd.conf** configuration file in a text editor. The location of this file depends on your installation. For example, it could be located in one of these places: • **/usr/local/apache/conf/httpd.conf** • **/etc/apache/httpd.conf**
2	Locate the following text: `<Directory "/var/apache/cgi-bin">`

3	After the `<Directory "/var/apache/cgi-bin">` line, add the following text:
	`ScriptAlias /ccp/ccp.jnlp "/IN/html/ccp/cgi-bin/ccp.jnlp"`
	`<Directory "/IN/html/ccp/cgi-bin">`
	`AllowOverride None`
	`Options None`
	`Order allow,deny`
	`Allow from all`
	`</Directory>`

4	Save and close the file.
5	Restart the apache daemon with either command, depending on where the **.conf** configuration file is located, for example: `/usr/apache/bin/apachect1 restart`

Multiple Customers

If multiple customers are using the same platform, you can start the CCP by using a separate JNLP file for each customer.

Creating a Customer JNLP File

Follow these steps to create a separate customer JNLP file.

Step	Action
1	Log in to the SMS as the root user.

Step	Action
2	Copy the **/IN/html/ccp/cgi-bin/ccp.jnlp** file and save it with a different name. **Example:** `cp ccp.jnlp customer.jnlp` Where *customer* is the customer name you want to use.
3	Open the customer file in a text editor. **Example:** `vi /IN/html/ccp/cgi-bin/customer.jnlp`
4	Add to the apache config: `ScriptAlias /ccp/customer.jnlp "/IN/html/ccp/cgi-bin/customer.jnlp"`
5	Save and close the file.

Chapter 3

Background Processes on the SMS

Overview

Introduction

This chapter provides a description of the programs or executables used by CCS as background processes on the SMS.

Executables are located in the **/IN/service_packages/CCS/bin** directory.

Some executables have accompanying scripts that run the executables after performing certain cleanup functions. All scripts should be located in the same directory as the executable.

For more information about the processes and systems that use these programs and executables, see *System Overview* (on page 1).

Warning: It is a prerequisite for managing these core service functions that the operator is familiar with the basics of Unix process scheduling and management. Specifically, the following Unix commands:

- init (and inittab)
- cron (and crontab)
- ps
- kill

In this chapter

This chapter contains the following topics.

CHECK_PC_DELETION 88
acsCompilerDaemon 88
ccsBeOrb 89
ccsCB10HRNAES 108
ccsCB10HRNSHA 108
ccsCDRFileGenerator 108
ccsCDRLoader 111
ccsCDRTrimDB 129
ccsCDRTrimFiles 130
ccsChangeDaemon 131
ccsExpiryMessageLoader 137
ccsExternalProcedureDaemon 141
ccsLegacyPIN 142
ccsPeriodicCCRecharge 142
ccsPeriodicCharge 144
ccsProfileDaemon 152
ccsReports 163
ccsWalletExpiry 167
libccsCommon 170
VoucherRedeemFail Files 170

CHECK_PC_DELETION

Overview

This procedure is run once a day through a script **/IN/service_packages/CCS/bin/ccs_pc_delete.sh** launched through the crontab of the ccs_oper user.

Do a `crontab -e` as ccs_oper to see the related entry in the crontab or to change the date of occurrence.

This procedure will fully delete any periodic charges that are ready for final deletion. This includes all references to the periodic charge in other tables and also all references to the associated balance type. Deletion of a periodic charge will include any references to the charge by the Subscriber Profile Manager.

Deletion criteria

A periodic charges that is ready for final deletion has the following criteria:

- DELETION_DATE is not null and is earlier than (<) sysdate.

acsCompilerDaemon

Purpose

The acsCompilerDaemon generates the fast-lookup binary compiled control plan data which is then used by the ACS service logic to process calls at execution time.

The acsCompilerDaemon runs continuously, polling the database to look for newly written control plans and control plan structures (for example, indicated by database field ACS_CALL PLAN.BUILD = B). It polls the database every “alertTimeout” seconds. Due to the way Oracle reacts to signals, signals are masked during the time the process is both waiting for an alert to occur and the time spent compiling control plans.

You need to configure acsCompilerDaemon for the CCS system to run successfully because CCS runs as an ACS service.

It is run by acs_oper in the **acs.conf** file.

For more information about ACS, control plans and the **acs.conf** file, see *ACS Technical Guide*.

Startup - nonclustered

In a non clustered environment this task is started automatically by entry acs0 in the inittab, through the **/IN/service_packages/ACS/bin/acsComplierDaemonStartup.sh** shell script.

You can check if the process is running by using the Unix ps command. We assume that you are familiar with Unix processes and with the Unix commands to manage them.

To check the process, enter:

```
ps -ef | grep acsCompilerDaemon
```

Result: If the acsCompilerDaemon is running, you should see output like the following:

```
acs 23857 23853 49 14:33:20 pts/5 0:00 acsCompilerDaemon
```

When ACS is installed, the startup inittab entry is added by the install process. The inittab entry waits until Oracle has started and then executes.

Startup - clustered

In a clustered environment this task is started automatically by the Sun Plex manager. The files required by the Sun Plex manager are located in the **/opt/ESERVAcsCompilerDaemon** directory.

This is configured by the acsCluster package, and will set up the use of the Sun Plex manager to start, stop, restart and move the failover processes to other nodes as required.

Location

This binary is located on the SMS node.

Parameters

The acsCompilerDaemon does not support any command line parameters; it is completely configured in the **acs.conf** file. For more information about the **acs.conf** file, see *ACS Technical Guide*.

Failure

If the acsCompilerDaemon has failed, then control plans will not be compiled. This can be detected by executing the following SQL statement on the SMF database instance:

```
SELECT ID from ACS_CALL_PLAN where BUILD='B';
```

Under normal operation, control plans will only remain in the B state for a few seconds at most.

Output

The acsCompilerDaemon writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/ACS/tmp/acsCompilerDaemon.log**.

ccsBeOrb

Purpose

The ccsBeOrb interface is responsible for updating and retrieving subscriber account states for other processes. Updates to an account are also made from this ORB interface to the other Voucher and Wallet Server in the pair.

About Configuring CORBA Connections for ccsBeOrb

The CorbaServices section in the **eserv.config** configuration file on the SMS node defines common connection parameters for CORBA services for ccsBeOrb. The CorbaServices configuration overrides the default and command-line values specified for CORBA listen ports and addresses. If you are using IP version 6 addresses, then you must include the CorbaServices section in the **eserv.config** file. However, this section is optional if you are using only IP version 4 addresses.

You configure the CorbaServices section of the **eserv.config** configuration file on the SMS by using the following syntax:

```
CorbaServices = {
    AddressInIOR = "hostname"
    ccsBeOrbListenPort = port
    OrbListenAddresses = [
        "ip_address1",
        "ip_address2",
]
```

```
}
```

Where:

- *hostname* is the hostname or IP address to place in the IOR (Interoperable Object Reference) for the CORBA service.
- *port* is the number of the port on which ccsBeOrb will listen. The `ccsBeOrbListenPort` parameter overrides the port number set by the `listenPort` parameter.
- *ip_address1,ip_address2* lists the IP addresses on which CORBA services listen for incoming requests. The list of IP addresses in the `OrbListenAddresses` parameter can include both IP version 6, and IP version 4 addresses. The `OrbListenAddresses` parameter overrides the IP address set by the `listenHost` parameter.

For more information about configuring CORBA services, see *Service Management System Technical Guide*.

Startup - non clustered

This task is started by entry ccs3 in the inittab, through the **/IN/service_packages/CCS/bin/ccsBeOrbStartup.sh** shell script.

You can check if the process is running by using the Unix ps command.

To check the process, enter:

```
ps -ef | grep ccsBeOrb
```

Result: The listed process is the compiler process.

ccsBeOrb Start-up for Operational Implementation

The ccsBeOrb process is started automatically by placing it in the Unix Initialization table, inittab.

To start the compiler manually, enter:

```
CCS_ROOT/bin/ccsBeORB
```

Result: Placing the ccsBeOrb startup script in the inittab file ensures that if ccsBeOrb should die, it will be automatically restarted by the operating system within a few seconds.

Startup - clustered

In a clustered environment this task is started automatically by the Sun Plex manager. The files required by the Sun Plex manager are located in the **/opt/ESERVCcsBeOrb** directory.

This is configured by the ccsCluster package, and will set up the use of the Sun Plex manager to start, stop, restart and move the failover processes to other nodes as required.

Location

This binary is located on the SMS node.

Restart

Under certain circumstances, it is desirable to restart ccsBeOrb in order to pick up some configuration changes with minimal interruption to service. The most likely reasons for this would be:

- A new ccsBeOrb program has been compiled and linked
- Configuration file **eserv.config** has been modified

In this case, you can use `ps` to determine the process ID of the ccsBeOrb process, and use `kill -TERM` to terminate the process.

Shutdown

To terminate the ccsBeOrb, use the Unix command ps to identify the process number and kill it manually. Or, you can use the provided shell script, kill_CCS_be_orb to simplify the task.

Configuration - eserv.config

ccsBeOrb is configured by the `ccsBeOrb` section of the **eserv.config** file. The structure of the section is shown below.

Note: For more information about the configuration for the BeClient provided by the libBeClientIF library, see *VWS Technical Guide*.

```
ccsBeOrb = {
    beLocationPlugin = "lib"
    oracleUserPass = "usr/pwd"
    clientName = "name"

    heartbeatPeriod = microsecs
    messageTimeoutSeconds = seconds
    maxOutstandingMessages = int
    reportPeriodSeconds = seconds
    connectionRetryTime = seconds

    plugins = [
        {
            config="confStr",
            library="lib",
            function="str"
        }
        [...]
    ]

    confStr = {
        plugin configuration
    }

    notEndActions = [
        {type="str", action="[ACK |NACK]"}
        [...]
    ]

    plugin configuration - see plug-in-specific config

    stateConversions = {
        <A|P|D|F|S|T> = "str"[,
        ...]
    }
    voucherStateConversions = {
        <A|F|R|C|D|H> = "str"[,
        ...]
    }
    namingServer = {
        host = "host",
        port = port,
        name = "str"
        addHostPrefix = true|false
    }
    billingEngines = [
        {
            id = id,
            primary = { ip="ip", port=port },
```

```
            secondary = { ip="ip", port=port }
        }
        [...]
    ]
}
```

eserv.config parameters

The ccsBeOrb supports the following parameters in the ccsBeOrb section of the **eserv.config** file.

Note: This configuration section is also used by the ccsVWARSExpiry service library.

`billingEngines`

Syntax:
```
billingEngines = [
     { id = int
      primary = { ip="ip", port=port },
      secondary = { ip="ip", port=port }
     }
     [...]
 ]
```

Description: Overrides connection details that `beLocationPlugin` (on page 135) obtains from the database.

Type: Parameter array.

Optionality: Optional (`beLocationPlugin` finds connection details if not set).

Allowed:

Default:

Notes: Identifies the Voucher and Wallet Servers and assigns their Internet connection details.

Example:
```
billingEngines = [
     { id = 1,
      primary = { ip="192.0.2.0", port=1500 },
      secondary = { ip="192.0.2.1", port=1500 }
     }
 ]
```

`id`

Syntax: `id = int`

Description: This unique identifier for this Voucher and Wallet Server configuration.

Type: Integer

Optionality: Required, if this section is used

Allowed:

Default:

Notes: This parameter is part of the `billingEngines` parameter array.

Example: `id = 1`

`ip`

Syntax: `ip = "ip"`

Description: The Internet Protocol (IP) address of the Voucher and Wallet Server.

Type: String

Optionality: Required

Allowed: IP version 4 (IPv4) addresses, IP version 6 (IPv6) addresses

Default: None

Notes: This parameter is part of either the primary, or the secondary parameter group of the `billingEngines` parameter array.
You can use the industry standard for omitting zeros when specifying IPv6 addresses.

Examples:
```
ip = "192.0.2.0"
ip = "2001:db8:0000:1050:0005:0600:300c:326b"
ip = "2001:db8:0:0:0:500:300a:326f"
ip = "2001:db8::c3"
```

port

Syntax: `port = port`

Description: The port number associated with the address of the Voucher and Wallet Server.

Type: Integer

Optionality: Required

Allowed:

Default: None

Notes: This parameter is part of either the primary or secondary parameter group of the `billingEngines` parameter array.

Example:
```
port = 1500
```

primary

Syntax: `primary = { ip="ip", port=port }`

Description: The `primary` parameter group defines the Internet Protocol (IP) address and associated port number of the primary Voucher and Wallet Server.

Type: Parameter array

Optionality: Required if this section is used

Allowed:

Default:

Notes: This parameter is part of the `billingEngines` parameter array.

Examples:
```
primary = { ip="192.0.2.0", port=1500 }
primary = { ip = "2001:db8:0000:1050:0005:0600:300c:326b",
port=1500 }
primary = {ip = "2001:db8:0:0:0:500:300a:326f", port=1500 }
primary = { ip = "2001:db8::c3", port=1500 }
```

secondary

Syntax: `secondary = { ip="ip", port=port }`

Description: The `secondary` parameter group defines the Internet Protocol (IP) address and associated port number of the secondary Voucher and Wallet Server.

Type: Array

Optionality: Required, if this section is used

Allowed:

Default:

Notes: This parameter is part of the `billingEngines` parameter array.

Examples:

```
secondary = { ip="192.0.2.1", port=1500 }
secondary = { ip = "2001:db8:0000:1050:0005:0600:300c:326b",
port=1500 ]
secondary = {ip = "2001:db8:0:0:0:500:300a:326f", port=1500
}
secondary = { ip = "2001:db8::c3", port=1500 }
```

broadcastOptions

Syntax:

```
broadcastOptions = {
    aggregateNAckCodes = [config]
}
```

Description: Name of configuration section for the BeClient Broadcast plug-in libclientBcast.

Type: Parameter array

Optionality:

Allowed:

Default:

Notes: libclientBcast is used by a range of processes which connect to the beServer, including:

- BeClient
- PIbeClient
- ccsBeOrb

For more information about libclientBcast, see `libclientBcast`.

Example:

```
broadcastOptions = {
    aggregateNAckCodes = [ ]
}
```

aggregateNAckCodes

Syntax:

```
aggregateNAckCodes = [
    "NVOU"
]
```

Description: When this parameter is set, the BeClient waits for a response from all the VWS pairs in use and filters the responses from the broadcast request using the configured NAck codes.

Type: Parameter array

Optionality:

Allowed: NVOU

Default:

Notes: When a voucher recharge request is broadcast, this ensures that all the available VWS pairs are checked for the required voucher before a voucher not found message is returned to the requesting process.

Example:

clientName

Syntax: `clientName = "name"`

Description: The unique client name of the process.

Type: String

Optionality: Required

Allowed: Must be unique.

Default: The host name of the local machine.

Notes: The server generates clientId from a hash of *str*.

If more than one client attempts to connect with the same name, then some connections will be lost.

This parameter is used by libBeClientIF.

Example: `clientName = "scpClient"`

connectionRetryTime

Syntax: `connectionRetryTime = seconds`

Description: The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.

Type: Integer

Optionality: Required

Allowed:

Default: 5

Notes: This parameter is used by libBeClientIF.

Example: `connectionRetryTime = 2`

heartbeatPeriod

Syntax: `heartbeatPeriod = microsecs`

Description: The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.

Type: Integer

Optionality: Required

Allowed:

0	Disable heartbeat detection.
positive integer	Heartbeat period.

Default: 3000000

Notes: 1 000 000 microseconds = 1 second.

If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair.

This parameter is used by libBeClientIF.

Example: `heartbeatPeriod = 10000000`

listenHost

Syntax: `listenHost = "hostname"`

Description: The name of the host or the IP address on which ccsBeOrb will listen for incoming CORBA requests. An empty string implies all addresses.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: If listenHost is not set, then it defaults to the IP address corresponding to the result of the hostname UNIX command. If both an IP version 4 (IPv4) and an IP version 6 (IPv6) address exists for the hostname, then the IPv6 address will be used.

Notes:

Example: `listenHost = ""`

listenPort

Syntax: `listenPort = port`

Description: The number of the port on which ccsBeOrb will listen for incoming CORBA requests.

Type: Integer

Optionality:

Allowed:

Default: 0

Notes: The default (listenPort = 0) sets a random port.

Example:
```
listenPort = 10024
```

maxOutstandingMessages

Syntax: `maxOutstandingMessages = num`

Description: The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.

Type: Integer

Optionality: Required

Allowed:

Default: If this parameter is not set, the maximum is unlimited.

Notes: If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls.

The messages are queued until the Voucher and Wallet Server has reduced its outstanding load.

This parameter is used by libBeClientIF.

Example:
```
maxOutstandingMessages = 100
```

mergeWalletsOptions

Syntax:
```
mergeWalletsOptions = {
    oracleLogin = "name/password"
    mergeBucketExpiryPolicy = "outcome"
    mergeWalletExpiryPolicy = "outcome"
    allowedSourceWalletStates = "states"
    mergeWalletsTriggers = ["MGW "]
}
```

Description: Configuration for the beClientIF plug-in.

Type: Parameter group

Optionality:

Allowed:

Default:

Notes:

Example:

allowedSourceWalletStates

Syntax: `allowedSourceWalletStates = "str[...]"`

Description: The states the source wallet must be in to allow it to be merged with another wallet.

Type: String

Optionality: Required

Allowed:

P	Pre-use
A	Active
D	Dormant
S	Suspended
F	Frozen
T	Terminated

Default: None

Notes: At least one state must be included, or all merged will be disallowed.

Example: `allowedSourceWalletStates = "PA"`

oracleLogin

Syntax: `oracleLogin = "usr/pwd"`

Description: The login details the BeClient should use to log in to the SMF database, when performing merge wallet functions.

Type: String

Optionality: Optional

Allowed:

Default: /

Notes:

Example: `oracleLogin = "smf/smf"`

mergeBucketExpiryPolicy

Syntax: `mergeBucketExpiryPolicy = "str"`

Description: Determines what happens when the source wallet and destination wallet have buckets of the same balance type.

Type: String

Optionality: Optional (default used if not set).

Allowed:

merge	Update the bucket in the destination wallet. The updated bucket will have the: • combined value of the two buckets, and • expiry of whichever bucket has the latest expiry date.
move	Create a new bucket in the destination wallet. The new bucket will have the same balance type, value and expiry date as the bucket from the source wallet.

Default: merge

Notes:

Example: `mergeBucketExpiryPolicy = "move"`

mergeWalletExpiryPolicy

Syntax: `mergeWalletExpiryPolicy = "str"`

Description: Determines the way expiry dates for merged wallets are managed.

Type: String

Optionality: Optional

Allowed:

best	The expiry date of the wallet with the most time left is used.
ignore	The expiry date of the source wallet is ignored.

Default: best

Notes:

Example:
```
mergeWalletExpiryPolicy = "best"
```

mergeWalletsTriggers

Syntax:
```
mergeWalletsTriggers = [
 "str [...]"
]
```

Description: Wallets of this type starts the merge wallets plug-in.

Type: Array of strings.

Optionality: Required

Allowed: MGW

Default: None

Notes: The syntax must be typed exactly as shown in the example.

Example:
```
mergeWalletsTriggers = ["MGW "]
```

messageTimeoutSeconds

Syntax: `messageTimeoutSeconds = seconds`

Description: The time that the client process will wait for the server to respond to a request.

Type: Integer

Units: Seconds

Optionality: Required

Allowed:

1-604800	Number of seconds to wait.
0	Do not time out.

Default: 2

Notes: After the specified number of seconds, the client process will generate an exception and discard the message associated with the request.

This parameter is used by libBeClientIF.

Example:
```
messageTimeoutSeconds = 2
```

namingServer

Syntax:
```
namingServer = {
    host = "hostName",
    port = portNumber,
    name = "clientName"
}
```

Description: Registers with smsNamingServer so that screens can find the ccsBeOrb service.

Type: Parameter group.

Optionality:

Allowed:
Default:
Notes:
Example:

addHostPrefix

Syntax: `addHostPrefix = true|false`

Description: Whether or not to add the hostname as a prefix to the BeClient name when connecting to the beServer.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:
- true — Add the prefix.
- false — Do not add the prefix.

Default: true

Notes: Must be set to true.

Example: `addHostPrefix = false`

host

Syntax: `host = "hostName"`

Description: The hostname of the machine ccsBeOrb is running on.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: localhost

Notes: The `host` parameter is part of the `namingServer` parameter group.

Example: `host = "produsms01"`

name

Syntax: `name = "clientName"`

Description: The name of the client.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: ccsBeClientOrb

Notes: The `name` parameter is provided for backwards compatibility with old screens.

The `name` parameter is part of the `namingServer` parameter group.

Example: `name = "ccsBeClientOrb"`

port

Syntax: `port = portNumber`

Description: The number of the port on which the client listens.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 5556

Notes:

Example:
```
port = 5556
```

notEndActions

Syntax:
```
notEndActions = [
    {type="str", action="[ACK|NACK]"}
    [...]
]
```

Description: The `notEndActions` parameter array is used to define the messages associated with dialogs that should not have their dialog closes, because the dialog is closed by default. This facilitates failover.

Type: Parameter array.

Optionality: Required

Allowed:

Default:

Notes: If the incoming dialog for a call closes and the last response received was of the notEndActions type, the client process sends an ABRT message. The ABRT message allows the VWS to remove the reservation. An example of this situation would be where slee_acs has stopped working.

This parameter is used by libBeClientIF.

For more information about slee_acs, see *ACS Technical Guide*.

Example:
```
notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

plugins

Syntax:
```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    ...
 ]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Optional (as plug-ins will not be loaded if they are not configured here, this parameter must include any plug-ins which are needed to supply application functions; for more information about which plug-ins to load, see the BeClient section for the application which provides the BeClient plug-ins).

Allowed:

Default: Empty (that is, do not load any plug-ins).

Notes: The libclientBcast plug-in must be placed last in the plug-ins configuration list.
For more information about the libclientBcast plug-in, see *VWS Technical Guide*.
This parameter is used by libBeClientIF.

Example:
```
plugins = [
    {
        config="broadcastOptions"
        library="libclientBcast.so"
        function="makeBroadcastPlugin"
    }
]
```

config

Syntax: `config="name"`
Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `config="voucherRechargeOptions"`

function

Syntax: `function="str"`
Description: The function the plug-in should perform.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `function="makeVoucherRechargePlugin"`

library

Syntax: `library="lib"`
Description: The filename of the plug-in library.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `library="libccsClientPlugins.so"`

Voucher and wallet plugins

There are four plug-ins which provide functionality for the PIbeClient:

1. Voucher recharge (VRW)
2. Voucher type recharge (VTR)

3 Merge wallets (MGW)
4 *Broadcast* (on page 106)

Note: The broadcast plug-in configuration must be placed last in the `plugins` configuration section.

Each plug-in can have a configuration section. The name of this subsection will match the string provided for the config parameter in the `plugins` subsection.

Example: The Voucher Recharge plug-in has config set to `voucherRechargeOptions`. So the configuration section for this plug-in is:

```
voucherRechargeOptions = {
    ...
}
```

reportPeriodSeconds

Syntax:	`reportPeriodSeconds = seconds`
Description:	The number of seconds separating reports of failed messages.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	
Default:	10
Notes:	BeClient issues a failed message report: • For timed-out messages • For unrequested responses • For new calls rejected because of congestion • For messages with invalid Voucher and Wallet Server identifiers • If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`. This parameter is used by libBeClientIF.
Example:	`reportPeriodSeconds = 10`

stateConversions

Syntax:	`stateConversions = {`
	`A = "ACTV",`
	`P = "PREU",`
	`D = "DORM"`
	`F = "FROZ",`
	`S = "SUSP",`
	`T = "TERM"`
	`H = "RSVD"`
	`}`

Description:	Converts from ESCHER encoding to a single character and back.
Type:	Array
Optionality:	
Allowed:	
Default:	
Notes:	
Example:	

voucherRechargeOptions

Syntax:

```
voucherRechargeOptions = {
    srasActivatesPreuseAccount = true|false
    voucherRechargeTriggers = [
        "VRW
    ]
    sendBadPin = true|false
}
```

Description: Configures the voucher recharge plug-in.

Type: Array

Optionality:

Allowed:

Default:

Notes:

Example:

sendBadPin

Syntax: `sendBadPin = true|false`

Description: Whether or not to increment the Bad PIN count for a failed voucher redeem.

Type: Boolean

Optionality: Optional

Allowed:

- true – Increment Bad PIN count for each failed attempt to recharge a voucher.
- false – Do not increment Bad PIN count for failed attempts to recharge a voucher.

Default: false

Notes: This parameter:

- applies only to an invalid voucher number or voucher PIN. It does not apply to failed wallet recharges
- is part of the `voucherRechargeOptions` parameter group

Example: `sendBadPin = false`

srasActivatesPreuseAccount

Syntax: `srasActivatesPreuseAccount = true|false`

Description: Sets whether or not alternate subscribers can activate subscriber accounts which are in a pre-use state.

Type: Boolean

Optionality: Optional

Allowed:

- true – A scratch card alternate subscriber can activate a pre-use account.
- false – A scratch card alternate subscriber cannot activate a pre-use account.

Default: true

Notes: This parameter is:

- Not used by ccsBeOrb
- Part of the `voucherRechargeOptions` parameter group

Example: `srasActivatesPreuseAccount = false`

voucherRechargeTriggers

Syntax:
```
voucherRechargeTriggers = [
    "VRW "
]
```
Description: This message triggers the voucher recharge plug-in.
Type: Array
Optionality: Required
Allowed: VRW
Default:
Notes: This parameter array is part of the `voucherRechargeOptions` parameter group.
Example:

voucherServerCacheLifetime

Syntax: `voucherServerCacheLifetime = seconds`
Description: Time in seconds to hold items in the voucher server ID cache.
Type: Integer
Optionality: Optional
Allowed: Any positive decimal integer.
Default: 600 (seconds)
Notes:
Example:
```
voucherServerCacheLifetime = 600
```

voucherServerCacheCleanupInterval

Syntax: `voucherServerCacheCleanupInterval = seconds`
Description: Time in seconds between purges of the voucher server id cache.
Type: Integer
Optionality: Optional
Allowed: Any positive decimal integer.
Default: 60 (seconds)
Notes:
Example:
```
voucherServerCacheCleanupInterval = 60
```

voucherTypeRechargeOptions

Syntax:
```
voucherTypeRechargeOptions = {
   srasActivatesPreuseAccount = true|false
   voucherTypeRechargeTriggers = ["VTR "]
}
```
Description: Configures the voucher type recharge plug-in.
Type: Parameter group.
Optionality:
Allowed:
Default:
Notes:
Example:

srasActivatesPreuseAccount

Syntax: `srasActivatesPreuseAccount = true|false`

Description: Sets whether or not alternate subscribers can activate subscriber accounts which are in a pre-use state.

Type: Boolean

Optionality: Optional

Allowed:
- true – A scratch card alternate subscriber can activate a pre-use account.
- false – A scratch card alternate subscriber cannot activate a pre-use account.

Default: true

Notes: This parameter is:
- Not used by ccsBeOrb
- Part of the `voucherRechargeOptions` parameter group

Example:
```
srasActivatesPreuseAccount = false
```

voucherTypeRechargeTriggers

Syntax:
```
voucherTypeRechargeTriggers = [
    str [...]"
]
```

Description: Starts the voucher type recharge plug-in.

Type: Array

Optionality: Required

Allowed: VRW

Default:

Notes: This parameter array is part of the `voucherTypeRechargeOptions` parameter group.

Example:
```
voucherTypeRechargeTriggers = ["VTR "]
```

voucherStateConversions

Syntax:
```
voucherStateConversions = {
    str = "ESCHER"[,
    ...]
}
```

Description: Converts from ESCHER encoding to a single character and back.

Type: Array

Optionality: Required.

Allowed:

Value	Description
A = "ACTV"	Active
F = "FRZN"	Frozen
R = "RDMD"	Redeemed
C = "CRTD"	Created
D = "DLTD"	Deleted
H = "RSVD"	Held

Default:

Notes:

Example:

```
voucherStateConversions = {
    A = "ACTV",
    F = "FRZN",
    R = "RDMD",
    C = "CRTD"
    D = "DLTD"
    H = "RSVD"
}
```

Broadcast plug-in

The Broadcast PIbeClient plug-in overrides the beLocationPlugin that would normally load connection details from the database.

The `plugins` section must include the following configuration to load this plug-in.

```
{
    config="",
    library="libccsClientPlugins.so",
    function="makeBroadcastPlugin"
}
```

Notes:

- This plug-in must be the last in the `plugins` subsection.
- This plug-in has no configuration.
- The broadcast plug-in is required by the VRW and VTR plug-ins.

Example eserv.config

Here is an example `ccsBeOrb` section of the `CCS` section of the **eserv.config**.

Usage:

```
ccsBeOrb = {
    listenHost = ""
    listenPort = 10024
    clientName = "usmsprod01-ccsBeOrb"
    heartbeatPeriod = 10000000
    maxOutstandingMessages = 100
    connectionRetryTime = 2
    requestTimeoutSeconds = 0
    plugins = [
        { # Voucher recharge (VRW) plugin
        config="voucherRechargeOptions",
        library="libccsClientPlugins.so",
        function="makeVoucherRechargePlugin"
        }
        { # Voucher Type recharge (VTR) plugin
        config="voucherTypeRechargeOptions",
        library="libccsClientPlugins.so"
        function"makeVoucherTypeRechargePlugin"
        }

        { # Merge Wallets plugin
        config="mergeWalletsOptions",
        library="libccsClientPlugins.so",
        function="makeMergeWalletsPlugin"
        }
        { # Broadcast plugin needed by VRW
        config="broadcastOptions",
        library="libclientBcast.so",
        function="makeBroadcastPlugin"
```

```
    }
]

broadcastOptions = {
    aggregateNAckCodes = [
        "NVOU"
    ]
}

voucherRechargeOptions = {
    srasActivatesPreuseAccount = false
    voucherRechargeTriggers = [
        "VRW "
    ]
    voucherServerCacheLifetime = 600
    voucherServerCacheCleanupInterval = 60
    sendBadPin = false
}

voucherTypeRechargeOptions = {
    srasActivatesPreuseAccount=false
    voucherTypeRechargeTriggers = ["VTR "]
}

mergeWalletsOptions = {
    oracleLogin = "/"
    mergeBucketExpiryPolicy = "merge"
    mergeWalletExpiryPolicy = "best"
    allowedSourceWalletStates = "PADS"
    mergeWalletsTriggers = ["MGW "]
}

notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
stateConversions = {
    A = "ACTV",
    P = "PREU",
    D = "DORM"
    F = "FROZ",
    S = "SUSP",
    T = "TERM"
}
voucherStateConversions = {
    A = "ACTV",
    F = "FRZN",
    R = "RDMD"
    C = "CRTD"
    D = "DLTD"
    H = "RSVD"
}
namingServer = {
    host = "usmsprod01",
    port = 5556,
    name = "ccsBeClientOrb"
}
billingEngines = [
    { id = 1,
```

```
          primary = { ip="190.0.2.0", port=1500 },
          secondary = { ip="190.0.2.1", port=1500 }
        }
    ]
}
```

Failure

If the ccsBeORB fails, updates to accounts will fail.

Output

The ccsBeORB writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsBeOrb.log**.

ccsCB10HRNAES

License

The ccsCB10HRNAES library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsCB10HRNSHA

License

The ccsCB10HRNSHA library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsCDRFileGenerator

Purpose

ccsCDRFileGenerator takes EDRs created through the CCS UI and writes them to a flat file equivalent. This file of EDRs is then get loaded into CCS_BE_CDR by ccsCDRLoader.

Startup - non clustered

This task is started by entry ccs7 in the inittab, through the **/IN/service_packages/CCS/bin/ccsCDRFileGeneratorStartup.sh** shell script.

Startup - clustered

In a clustered environment this task is started automatically by the Sun Plex manager. The files required by the Sun Plex manager are located in the **/opt/ESERVCcsCDRFileGenerator** directory**.**

This is configured by the ccsCluster package, and will set up the use of the Sun Plex manager to start, stop, restart and move the failover processes to other nodes as required.

Parameters

The `ccsCDRFileGenerator` section includes the following parameters from the `CCS` section of **eserv.config**:

Usage:

```
ccsCDRFileGenerator = {
    OutputDirectory = "/IN/service_packages/CCS/logs/CDR"
    BaseName = "ccsCDRFileGenerator"
    OracleUsernamePassword = "smf/smf"
    SleepDuration = 60
    BillingEngineID = 0
    SCPID = 0
}
```

The available parameters are:

`BaseName`

Syntax:	`BaseName = "`*`name`*`"`
Description:	Base name of the output files
Type:	String
Optionality:	Mandatory
Allowed:	
Default:	None
Notes:	
Example:	

`BillingEngineID`

Syntax:	`BillingEngineID =` *`id`*
Description:	Billing Engine ID.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	0
Notes:	This should not match any actually installed BEID.
Example:	

`OracleUsernamePassword`

Syntax:	`OracleUsernamePassword = "`*`usr/pwd`*`"`
Description:	Username and password used to connect to SMF database.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	"/"
Notes:	
Example:	

`OutputDirectory`

Syntax: `OutputDirectory = "dir"`
Description: Directory name where ccsCDRFileGenerator will write output files.
Type: String
Optionality: Mandatory
Allowed:
Default: None
Notes:
Example:

`SCPID`

Syntax: `SCPID = id`
Description: ID of the SLC.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 0
Notes:
Example:

`SleepDuration`

Syntax: `SleepDuration = secs`
Description: The number of seconds ccsCDRFileGenerator will pause before generating a new file.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 60
Notes:
Example:

`TempOutputDirectory`

Syntax: `TempOutputDirectory = "dir"`
Description: The directory where the temporary files will be generated.
Type: String
Optionality: Mandatory
Allowed:
Default: None
Notes:
Example:

Failure

If ccsCDRFileGenerator fails, any EDRs generated through the CCS UI will fail.

Output

The ccsCDRFileGenerator writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsCDRFileGenerator.log**.

ccsCDRLoader

Purpose

The EDR loader (ccsCDRLoader) process periodically scans its input directory for EDR files. To the information it finds in these files, the process adds extra information derived from its plug-in libraries. It then writes the lot to the CCS_BE_CDR table in the database.

Some customers want to retain event data records outside the NCC system. The FileWriterCDRLoaderPlugin therefore rewrites each EDR so that it contains the same information as the database. Rewritten EDRs are placed in an output directory. See overview *Diagram* (on page 44).

Reprocessing Failed EDRs

If a CDR loader plug-in fails to process a particular EDR, then the ccsCDRLoader process carries out the following actions:

1. Save the EDR to a file for reprocessing. Any processing changes prior to the plug-in that failed are retained.
2. Add a special FAILED_PLUGIN tag holding the name of the plug-in which failed to the EDR.
3. Report the plug-in error in the log file.

When reprocessing EDRs, the ccsCdrLoader carries out the following actions:

1. If it finds an EDR that contains the FAILED_PLUGIN tag, then it iterates through the plug-in list until it finds the plug-in held in the FAILED_PLUGIN tag.
2. ccsCDRLoader then processes the EDR starting from the failed plug-in.

Note: You configure the location and maximum size of files that contain the failed EDRs by setting the *errDir* (on page 117) and *maxPluginFailFileSize* (on page 118) parameters in **eserv.config**.

Oracle Configuration

ccsCDRLoader requires an SMF entry in the Oracle file **tnsnames.ora**. The entry should be in the following format:

```
SMF =
    (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP) (HOST = hostname) (PORT = 1521))
    (CONNECT_DATA = (SID = SMF)))
```

Where:

hostname is the host name of the SMS machine.

If required, this entry may be modified depending on the individual platform and connection requirements.

Note: For more information about configuring **tnsnames.ora**, see *Oracle Net8 Admin Guide*.

Startup - Non Clustered

In a non clustered installation the ccsCDRLoader is started by an entry in the inittab, through the **/IN/service_packages/CCS/bin/ccsCDRLoaderStartup.sh** shell script.

Multiple Loaders

To allow multiple instances of the CDRLoader to run in parallel, two environment variables can be specified in the startup script for each CDRLoader to override the **eserv.config** parameters, for example:

Script 1:

```
 /IN/service_packages/CCS/.profile-sms
CCSCDRLOADER_INDIR=/IN/service_packages/CCS/tmp/CDR1/CDR-in
export CCSCDRLOADER_INDIR
CCSCDRLOADER_OUTDIR=/IN/service_packages/CCS/tmp/CDR1/CDR-store
export CCSCDRLOADER_OUTDIR
exec /IN/service_packages/CCS/bin/ccsCDRLoader
```

Script 2:

```
 /IN/service_packages/CCS/.profile-sms
CCSCDRLOADER_INDIR=/IN/service_packages/CCS/tmp/CDR2/CDR-in
export CCSCDRLOADER_INDIR
CCSCDRLOADER_OUTDIR=/IN/service_packages/CCS/tmp/CDR2/CDR-store
export CCSCDRLOADER_OUTDIR
exec /IN/service_packages/CCS/bin/ccsCDRLoader
```

Note: To define a TZ that the NOTICE messages by ccsCDRLoader are logged in, add DEBUG_TZ environment variable in the **ccsCDRLoaderStartupStartup.sh** script before the exec statement. For example:
`export DEBUG_TZ=Asia/Kolkata`

Startup - clustered

In a clustered environment the ccsCDRLoader is started automatically by the Sun Plex manager. The files required by the Sun Plex manager are located in the **/opt/ESERVCcsCDRLoader** directory.

This is configured by the ccsCluster package at installation, and will set up the use of the Sun Plex manager to start, stop, restart and move the failover processes to other nodes as required.

ccsCDRLoader Command Line Parameters

The ccsCDRLoader process supports the following optional command line parameters:

`ccsCDRLoader [--vwars_range` *`vwars_num`*`[-`*`vwars_num`*`] [--serverID` *`ID`*`] [--inDir` *`dir`*`] [--outDir` *`dir`*`]`

Where:

- `--vwars_range` *`vwars_num`*`[-`*`vwars_num`*`]` specifies the beVWARS number or number range for which EDR files will be processed. You must specify non-negative numbers for the *vwars_num* values.

  The EDR filenames must include the string "beVWARS-*"* followed by a beVWARS number within the specified range*.* EDR files with filenames that do not include this string are skipped. In addition, EDRs that are generated by the ccsCDRFileGenerator process will be matched only if the ccsCDRFileGenerator output filename contains the "beVWARS-" string. You configure the ccsCDRFileGenerator output filename to include this string by setting its *BaseName* (on page 109) parameter.

  Example syntax: `ccsCDRLoader --vwars_range 0-2`
- `--serverID` *`ID`* (string) specifies the unique server ID for the input files that will be processed by this instance of ccsCDRLoader. ccsCDRLoader matches the server ID against any part of the EDR input filename, and not just the hostname of the server that generated the EDR file.

  Example syntax: `ccsCDRLoader --serverID vws01`
- `--inDir` *`dir`* specifies the path and directory location of the input files. This value overrides the input directory configured for ccsCDRLoader in the **eserv.config** configuration file.

  Example syntax: `ccsCDRLoader --inDir /IN/service_packages/CCS/logs/CDR-in`

- `--outDir dir` specifies the path and directory location of the output files. This value overrides the output directory configured for ccsCDRLoader in the **eserv.config** configuration file.
  Example syntax: `ccsCDRLoader --outDir /IN/service_packages/CCS/logs/CDR-store/vws-0-2`

ccsCDRLoader Plug-in Libraries

The ccsCDRLoader can be extended by installing plug-in libraries. This section lists the ccsCDRLoader plug-in libraries that are available as a standard. Other plug-in libraries may also be installed as required.

The plug-ins are included in the *pluginLibs* (on page 118) array.

AcsCustIdPlugin

This plug-in library checks the EDR for the presence of the ACS_CUST_ID tag. If it is not present, the plug-in looks up ACS_CUST_ID in the ACS_ACCT table on the VWS, using the ACCT_ID tag from the EDR to identify the correct record in the table.

This function is contained within the libAcsCustIdPlugin.so library, and is used if this library is referenced within the *pluginLibs* (on page 118) array.

Note: This plug-in library does not accept any parameters.

AcctHistPlugin

This plug-in library updates the CCS_ACCT_HIST_INFO table with account details, such as expiry date, when processing relevant EDRs.

This function is contained within the **libAcctHistPlugin.so** library.

See AcctHistPlugin Parameters for configuration details.

CDRStoreDBPlugin

This plug-in library updates the CCS_BE_CDR table with EDR details.

This function is contained within the **libCDRStoreDBPlugin.so**.

Note: This plug-in library does not accept any parameters.

CreditCardDetailsPlugin

This plug-in library, for CC_Recharge EDRs (type 9), updates the CCS_CREDIT_CARD_DETAILS table with the last recharge date.

This function is contained within the **libCreditCardDetailsPlugin.so** library.

Note: This plug-in library does not accept any parameters.

FileWriterCDRLoaderPlugin

The plug-in has two functions:

- FileWriterCDRLoaderPlugin rewrites each EDR file with the same information that CDRStoreDBPlugin writes to the database.

  After the EDR loader process reads an EDR file, other EDR loader plug-in libraries may add extra information. All of this information is then written to the database. Some customers want to extract event data records from the Oracle system and retain them elsewhere. The FileWriterCDRLoaderPlugin therefore rewrites each EDR so that it contains the same

information as the database. Rewritten EDRs are placed in an output directory.

- Optionally, FileWriterCDRLoaderPlugin converts time events recorded in the EDR source files to the equivalent time in a configured time zone. It uses the converted time events when it rewrites the EDRs.

  The Oracle system manages all time events as if they occurred in the Coordinated Universal Time (UTC) zone. When an EDR file is rewritten, you can have FileWriterCDRLoaderPlugin use a different time zone. To do that you set up FileWriterCDRLoaderPlugin's cdrTimeZone configuration parameter. If cdrTimeZone is not configured or is configured incorrectly, time events will be written for the UTC time zone.

MsisdnCDRLoaderPlugin

This plug-in is optionally loaded based on the presence of the **libMsisdnCDRLoaderPlugin.so** in the *pluginLibs* (on page 118) configuration array.

The purpose of this plug-in is to look up the MSISDN (CLI) corresponding to the ACCT_REF_ID in the EDR tag. For type 3 (expiration) EDRs, if the ACCT_REF_ID is '0' then the ACCT_ID field will be used to look up the MSISDN instead.

MsisdnCDRLoaderPlugin is configured in the *MsisdnCDRLoader* (on page 128) section of **eserv.config**.

RechargeSMSPlugin

This plug-in library sends notifications to subscribers after a recharge, for the following EDR types:

- Recharge
- Freeform Recharge
- CC Recharge
- Voucher Freeform Recharge

This function is contained within the **libRechargeSMSPlugin.so** library.

See RechargeSMSPlugin Parameters for configuration details.

VoucherRedeemFailPlugin

This plug-in is optionally loaded based on the presence of the **libVoucherRedeemFailPlugin.so** in the *pluginLibs* (on page 118) configuration array.

The purpose of this plug-in is to trap and report on all non successful EDR type 15 records.

VoucherRedeemFailPlugin is configured in the *VoucherRedeemFail* (on page 122) section of **eserv.config**.

Part of the reporting is the inclusion of the MSISDN, which is only available when the **libMsisdnCDRLoaderPlugin.so** is loaded. To ensure this, **eserv.config** must have the **libMsisdnCDRLoaderPlugin.so** plug-in entry before this **libVoucherRedeemFailPlugin.so** plug-in entry on the *pluginLibs* (on page 118) array.

VoucherRedeemPlugin

This plug-in library, for recharge EDRs, updates the CCS_VOUCHER_REFERENCE table with the account reference id and redemption date.

This function is contained within the libVoucherRedeemPlugin.so library and is only required if the Voucher Management module is installed.

This plug-in library is configured in the *voucherRedeemPlugin* (on page 121) section of **eserv.config**.

CDR Loader Plug-in Parameters

The ccsCDRLoader process, and its plug-ins, are configured by the parameters in the ccsCDRLoader section of the **eserv.config** file.

CDR Loader Configuration Example

The following configuration shows example configuration for the ccsCDRLoader process in the **eserv.config** file.

```
ccsCDRLoader = {
    inDir = "/IN/service_packages/CCS/logs/CDR-in"
    inDirType = "HASH"
    outDir = "/IN/service_packages/CCS/logs/CDR-store"
    outDirType = "HASH"
    outDirExpectedFiles = 65536
    outDirBucketSize = 128
    readAheadNumFiles = 25
    cdrBufferSize = 4096
    scanInterval = 1
    statisticsInterval = 60
    loadZeroLenthCalls = true
    dbUserPass = "/"
    suffixToIgnore = ".tmp"
    commitInterval = 500
    filePocessing = "DELETE"
    maxPluginFailFileSize = 5000
    errDir = "/IN/service_packages/CCS/logs/CDR-err"
    pluginLibs = [
        "libAcsCustIdPlugin.so"
        "libVoucherRedeemPlugin.so"
        "libAcctHistPlugin.so"
        "libCreditCardDetailsPlugin.so"
        "libCDRStoreDBPlugin.so"
        "libFileWriterCDRLoaderPlugin.so"
        "libResetWaitForRechargePlugin.so"
        "libMsisdnCDRLoaderPlugin.so"
        "libVoucherRedeemFailPlugin.so"
    ]

    VoucherRedeemFail = {
        tempReportDirectory = "/IN/service_packages/CCS/tmp"
        archiveDirectory = "/IN/service_packages/CCS/logs/voucherRedeemFail"
        maxEDRs = 2000
        maxOpenDuration = 300
    }

    voucherRedeemPlugin = {
        useVoucherRedeemCDR = true
        additionalCdrTypes = [95,96]
    }

    AcctHistPlugin = {
        prodTypeSwapEventClass = "Product Type"
        prodTypeSwapEventName = "Product Type Swap"
        reasonChangeConfig = "/IN/service_packages/CCS/etc/changeReason.conf"
            acsCustomerIdData = [
                {
                    acsCustomerId = 1
                    promoCascade = "NE Test Promo Cascade"
                }
            ]
```

```
    }

    FileWriterCDRLoaderPlugin = {
        cdrTimeZone = "EST"
        ccsCDRFieldsTZ = [
            "RECORD_DATE"
            "TCS"
            "TCE"
            "ACTIVATION_DATE"
        ]
    }
}
```

CDR Loader Parameters

The ccsCDRLoader process supports the following parameters in the ccsCDRLoader section of the **eserv.config** file:

cdrBufferSize

Syntax:	`cdrBufferSize = int`
Description:	The size of the cache used by ccsCDRLoader and FileWriterCDRLoaderPlugin.
Type:	Integer
Units:	Kilobyte
Optionality:	Optional
Allowed:	
Default:	`2048`
Notes:	If you set `readAheadNumFiles` to be greater than 0 (zero), then set `cdrBufferSize` to a value that is large enough to cache input files by using the following formula: `cdrBufferSize` = ((*average_busy_period_input_file_size* multiplied by *`readAheadNumFiles`*) plus *buffer*) For example; if the average input file is 180 kilobytes and `readAheadNumFiles` is set to 20, then cdrBufferSize should be set to 4096. (cdrBufferSize = ((180 * 20) + 500) = 4100K)
Example:	`cdrBufferSize = 4096`

commitInterval

Syntax:	`commitInterval = num`
Description:	The number of EDRs to process before writing them to the database.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	200
Notes:	
Example:	`commitInterval = 200`

dbUserPass

Syntax:	`dbUserPass = "name/password"`
Description:	Contains the user name and password required to log on to the database.
Type:	String
Optionality:	Optional

Allowed:
Default: "/"
Notes:
Example: `dbUserPass = "/"`

errDir

Syntax: `errDir = "dir"`
Description: The path for the directory where the files containing EDRs which have failed due to a plug-in problem will be moved.
Type: String
Optionality: Optional (default used if not set).
Allowed: The directory path for an existing directory.
Default: "/IN/service_packages/CCS/logs/CDR"
Notes:
Example: `errDir = "/IN/service_packages/CCS/logs/CDR-err"`

fileProcessing

Syntax: `fileProcessing = "type"`
Description: Determines the file process.
Type: String
Optionality: Optional
Allowed:

DELETE	Time zone conversion is enabled.
MOVE	Time zone conversion is disabled

Default: "MOVE"
Notes: The time conversion feature of FileWriterCDRLoaderPlugin is affected by the fileProcessing parameter.
Example: `fileProcessing = "DELETE"`

inDir

Syntax: `inDir = "dir"`
Description: The directory from which EDRs are read.
Type: String
Optionality: Optional (default used if not set)
Allowed:
Default: "/IN/service_packages/CCS/logs/CDR/in"
Notes:
Example: `inDir = "/IN/service_packages/CCS/logs/CDR-in"`

inDirType

Syntax: `inDirType = "storeType"`
Description: Determines whether the input directory will be treated as a flat file store or a hash file store.
Type: String
Optionality: Optional

Allowed:	`FLAT`	Sub-directories are not searched.
	`HASH`	All files, including those in sub-directories, are processed.
Default:	"FLAT"	
Notes:	Can be set to `HASH` even if the directory is a flat file store, but not the other way around.	
Example:	`inDirType = "FLAT"`	

loadZeroLengthCalls

Syntax:	`loadZeroLengthCalls = true|false`
Description:	Defines whether zero-duration calls will be processed or skipped.
Type:	Boolean
Optionality:	Optional
Allowed:	`true` Zero-duration calls are processed. `false` Zero-duration calls are skipped.
Default:	true
Notes:	
Example:	`loadZeroLengthCalls = true`

maxPluginFailFileSize

Syntax:	`maxPluginFailFileSize = size`
Description:	The maximum size in KBs for files containing EDRs that have failed to process due to a plug-in problem. When a file containing failed EDRs reaches the maximum size, it is zipped and archived.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	A numeric value.
Default:	0 (zero)
Notes:	If the default is used then the file will not be archived.
Example:	`maxPluginFailFileSize = 5000`

pluginLibs

Syntax:	`pluginLibs = [`
	`  "1stLibrary"`
	`  "2ndLibrary"`
	`  …`
	`  …`
	`  …`
	`  "nthLibrary"`
	`]`

Description:	List of plug-in libraries to load.
Type:	Parameter array.
Optionality:	Optional
Allowed:	
Default:	`pluginLibs = [` `]`
Notes:	

Example:
```
pluginLibs = [
    "libAcsCustIdPlugin.so"
    "libVoucherRedeemPlugin.so"
    "libAcctHistPlugin.so"
    "libCreditCardDetailsPlugin.so"
    "libCDRStoreDBPlugin.so"
    "libFileWriterCDRLoaderPlugin.so"
]
```

outDir

Syntax: `outDir = "dir"`

Description: The directory to which EDRs are moved after they have been processed.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "/IN/service_packages/CCS/logs/CDR/out"

Notes:

Example:
```
outDir = "/IN/service_packages/CCS/logs/CDR-store"
```

outDirBucketSize

Syntax: `outDirBucketSize = filesPerLeaf`

Description: The number of files per leaf directory when the output directory contains the number of files specified by the `outDirExpectedFiles` parameter.

Type: Integer

Optionality: Optional

Allowed:

Default: 10

Notes: This parameter is ignored if `outDirType = "FLAT"`.

Example:
```
outDirBucketSize = 128
```

outDirExpectedFiles

Syntax: `outDirExpectedFiles = numberOfFiles`

Description: The number of EDR files expected in the directory defined by the `outDir` parameter.

Type: Integer

Optionality: Optional

Allowed:

Default:
```
outDirExpectedFiles = 100000
```

Notes: If `outDirType = "FLAT"`, this parameter is ignored.

Example:
```
outDirExpectedFiles = 65536
```

outDirType

Syntax: `outDirType = "storeType"`

Description: Sets the structure of the output directory defined by the `outDir` parameter.

Type: String

Optionality: Optional

Allowed: May be either `FLAT` or `HASH`.

Default: "FLAT"

Notes:

Example: `outDirType = "FLAT"`

readAheadNumFiles

Syntax: `readAheadNumFiles = int`

Description: Sets the maximum number of EDR input files to load into cache per `scanInterval`. When set to 0 (zero), the ccsCDRLoader queues all the EDR input files in the `inDir` directory for processing. The processed files are moved to the `outDir` directory only after ccsCDRLoader has finished loading all of them.

Type: Integer

Optionality: Optional (default used if not set)

Allowed: 0 or a positive integer

Default: 0

Notes: When you set `readAheadNumFiles` to a value that is greater than zero, then the recommended value for the `scanInterval` (on page 120) parameter is 1 (one). This ensures timely processing of the input files.

Example: `readAheadNumFiles = 25`

scanInterval

Syntax: `scanInterval = secs`

Description: The number of seconds between scans of the directory specified in the *inDir* (on page 117) parameter.

Type: Integer

Units: Seconds

Optionality: Optional

Allowed:

Default: 600

Notes:

- If the time taken to process the EDR input files is longer than the number of seconds specified for `scanInterval`, then the next scan occurs after processing has finished.
- If you expect the queue of EDR input files to be large, then to prevent input file backlogs, set `scanInterval` to a low value; for example, 1.

Example: `scanInterval = 1`

statisticsInterval

Syntax: `statisticsInterval = seconds`

Description: The number of seconds between statistical output.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: Defaults to the value set for the `scanInterval` parameter.

Notes: When set to:

- Less than or equal to `scanInterval`, statistics are output on every scan
- Greater than `scanInterval`, statistics are output following the next scan after `statisticsInterval` has expired

Example: `statisticsInterval = 60`

suffixToIgnore

Syntax: `suffixToIgnore = "suffix"`

Description: The suffix of files in the CDR in directory that should be ignored.

Type: String

Optionality: Optional.

Allowed:

Default: ".tmp"

Notes: For CDR files larger than the internal buffer size, ensures ccsCDRLoader is prevented from processing temporary files until the whole source CDR file has been processed.

Example: `suffixToIgnore = ".tmp"`

voucherRedeemPlugin

Syntax:
```
voucherRedeemPlugin = {
 useVoucherRedeemCDR = true|false
 additionalCdrTypes = [cdr_type]
 }
```

Description: The configuration for *VoucherRedeemPlugin* (on page 114) plug-in.

Type:

Optionality:

Allowed:

Default:

Notes:

Example:
```
voucherRedeemPlugin = {
 useVoucherRedeemCDR = true
 additionalCdrTypes = [95,96]
 }
```

additionalCdrTypes

Syntax: `additionalCdrTypes = [cdr_type]`

Description: Allows additional CDR types to be added to the REDEEMED_DATE column of the BE_VOUCHER table.

Type: Array

Optionality: Optional

Allowed: A CDR type greater than 66 as per MAX value in **ccsCDR.txt**

Default: Empty

Notes:

Example: `additionalCdrTypes = [95,96]`

useVoucherRedeemCDR

Syntax: `useVoucherRedeemCDR = true|false`

Description: Indicates that the Voucher Redeem CDR should be used instead of the Recharge CDR.

Type: Boolean

Optionality: Optional (default used if not set).
Allowed:
Default: false
Notes: Needed for split billing environments.
Example: `voucherRedeemCDR = true`

VoucherRedeemFail

Syntax:
```
VoucherRedeemFail = {
 parameters
}
```
Description: Configuration for the *VoucherRedeemFailPlugin* (on page 114) plug-in.
Type:
Optionality: Optional (defaults used if not present).
Allowed:
Default:
Notes:
Example:

archiveDirectory

Syntax: `archiveDirectory = "dir"`
Description: The location of the redeemed fail EDR file.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: "/IN/service_packages/CCS/logs/voucherRedeemFail"
Notes: This directory and *tempReportDirectory* (on page 123) should be in the same file system otherwise archiving will fail.
Example:
```
archiveDirectory =
"/IN/service_packages/CCS/logs/voucherRedeemFail"
```

maxEDRs

Syntax: `maxEDRs = num`
Description: The maximum number of EDR records in the file.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default: 2000
Notes:
Example: `maxEDRs = 3000`

maxOpenDuration

Syntax: `maxOpenDuration = seconds`
Description: The maximum amount of time in seconds the report file will be kept open.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:

Default: 300

Notes:

Example: `maxOpenDuration = 500`

tempReportDirectory

Syntax: `tempReportDirectory = "dir"`

Description: The directory where temporary report with failed voucher redeem records is stored.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "/IN/service_packages/CCS/tmp"

Notes: This directory and *archiveDirectory* (on page 122) should be in the same file system otherwise archiving will fail.

Example: `tempReportDirectory = "/IN/service_packages/CCS/tmp"`

AcctHistPlugin

Syntax:

```
AcctHistPlugin = {
 parameters
}
```

Description: Configures the account history plug-in.

Type: Parameter group.

Optionality:

Allowed:

Default:

Notes:

Example:

```
AcctHistPlugin = {
 prodTypeSwapEventClass = "Product Type"
 prodTypeSwapEventName = "Product Type Swap"
 reasonChangeConfig =
"/IN/service_packages/CCS/
 etc/changeReason.conf"
 acsCustomerIdData = [
 {
 acsCustomerId = 1
 promoCascade = "NE Test Promo Cascade"
 }
 ]
}
```

acsCustomerIdData

Syntax:

```
acsCustomerIdData = [
    {
        acsCustomerId = 1stIdentifier
        promoCascade = "1stName"
    }
    {
        acsCustomerId = 2ndIdentifier
        promoCascade = "2ndName"
    }
    …
    …
    …
    {
        acsCustomerId = nthIdentifier
        promoCascade = "nthName"
    }
]
```

Description: Lists data specific to each ACS customer ID.

Type: Parameter array.

Optionality: Optional

Allowed:

Default:

Notes: This parameter array is part of the `AcctHistPlugin` parameter group.

Example:

acsCustomerId

Syntax: `acsCustomerId = identifier`

Description: The number identifying the customer to whom this set of balances applies.

Type: Integer

Optionality: Mandatory

Allowed:

Default:

Notes: This parameter is part of the `acsCustomerIdData` parameter array.

Example: `acsCustomerId = 1`

promoCascade

Syntax: `promoCascade = "name"`

Description: The name of the promotional cascade that is saved in the `CASCADE` field of the EDR.

Type: String

Optionality: Mandatory

Allowed: This value must match an entry name in the Balance Type Cascades list, see *Charging Control Services User's Guide*, *Balance Type Cascades* topic.

Default: None

Notes: This parameter is part of the `acsCustomerIdData` parameter array.

Example: `promoCascade = "NE Test Promo Cascade"`

prodTypeSwapEventClass

Syntax: `prodTypeSwapEventClass = "class"`

Description: The content of the EVENT_CLASS field of product type swap EDRs.

Type: String

Optionality: Optional

Allowed:

Default: "Product Type"

Notes: This parameter is part of the `AcctHistPlugin` parameter group.

Example:
```
prodTypeSwapEventClass = "Product Type"
```

prodTypeSwapEventName

Syntax: `prodTypeSwapEventName = "name"`

Description: The content of the EVENT_NAME field of product type swap EDRs.

Type: String

Optionality: Optional

Allowed:

Default: "Product Type Swap"

Notes: This parameter is part of the `AcctHistPlugin` parameter group.

Example:
```
prodTypeSwapEventName = "Product Type Swap"
```

reasonChangeConfig

Syntax: `reasonChangeConfig = "dir"`

Description: The path to, and name of, the reason change configuration file.

Type: String

Optionality:

Allowed:

Default:

Notes:

- **changeReason.conf** lists available state changes and reasons for the changes. Information listed in is arranged in the following format:
  OldState;NewState;Reason
  For example:
  ```
  A;D;Active to Dormant
  D;A;Dormant to Active
  P;A;Active from Pre-Use
  ```
  The maximum reason length is 24 characters. If a longer reason is specified it will be truncated.
- This parameter is part of the `AcctHistPlugin` parameter group.

Example:
```
reasonChangeConfig = "/IN/service_packages/
CCS/etc/changeReason.conf"
```

RechargeSMSPlugin

Syntax:
```
RechargeSMSPlugin = {
 parameters
}
```

Description: Configuration for the recharge SMS plug-in.

Type: Parameter group.

Optionality: Optional

Allowed:
Default:
Notes:
Example:

smsFifoName

Syntax: `smsFifoName = "dir"`
Description: The path to and the name of the FIFO file to which SMS requests are written.
Type: String
Optionality: Optional
Allowed:
Default: "/tmp/ccsSSMRequest.fifo"
Notes: This parameter is part of the `RechargeSMSPlugin` parameter array.
Example: `smsFifoName = "/tmp/ccsSSMRequest.fifo"`

smsQueueSize

Syntax: `smsQueueSize = num`
Description: The maximum number of short messages to buffer.
Type: Integer
Optionality: Optional
Allowed:
Default: 1000
Notes: This parameter is part of the `RechargeSMSPlugin` parameter array.
Example: `smsQueueSize = 1000`

smsTTL

Syntax: `smsTTL = seconds`
Description: The maximum time that messages will be buffered.
Type: Integer
Units: Seconds
Optionality: Optional
Allowed:
Default: 600
Notes: This parameter is part of the `RechargeSMSPlugin` parameter array.
Example: `smsTTL = 600`

FileWriterCDRLoaderPlugin

Syntax:
```
FileWriterCDRLoaderPlugin = {
 cdrTimeZone = "zone"
 ccsCDRFieldsTZ = [
 "1stTag"
 "2ndTag"
 .
 .
 .
 "nthTag"
 ]
}
```

Description: Configuration for the file writer plug-in.

Type: Parameter group.

Optionality:

Allowed:

Default:

Notes:

Example:

ccsCDRFieldsTZ

Syntax:
```
ccsCDRFieldsTZ = [
 "1stTag"
 "2ndTag"
 .
 .
 .
 "nthTag"
]
```

Description: The time event field in the EDR file that will be converted to the time zone defined by the `cdrTimeZone` parameter.

Type: Array

Optionality:

Allowed:

Default:

Notes: This parameter is part of the `FileWriterCDRLoaderPlugin` parameter group.

Example:
```
ccsCDRFieldsTZ = [
    "RECORD DATE"
    "TCS"
    "TCE"
    "ACTIVATION DATE"
    "NEW_ACCT_EXPIRY"
    "NEW_BALANCE_EXPIRES"
    "OLD_ACCT_EXPIRY"
    "OLD_BALANCE_EXPIRES"
]
```

cdrTimeZone

Syntax:
```
cdrTimeZone = "tz"
```

Description: The time zone for time events written to EDR files.

Type: String

Optionality: Mandatory

Allowed: A UNIX time zone name.

Default:

Notes:
- You can see UNIX time zone names in the `/usr/share/lib/zoneinfo` directory. Type `ls` to see the high-level time zones. To see the sub-zones for say Asia, enter `ls Asia/`
- This parameter is part of the `FileWriterCDRLoaderPlugin` parameter group.

Example:
```
cdrTimeZone = "Dubai"
```

MsisdnCDRLoader

Syntax:
```
MsisdnCDRLoader = {
 parameters
}
```

Description: Configuration for the msisdn plug-in.

Example:
```
MsisdnCDRLoader = {
 CopyCliToMsisdn = true
 CopyCliToMsisdnRegExp =
"(\\|CDR_TYPE=13\\|\\|SERVICE=WIFI$|\\|SERVICE=WIFI|\\|)"
}
```

CopyCliToMsisdn

Syntax: `CopyCliToMsisdn = true|false`

Description: Sets whether or not to copy the CLI value to the MSISDN tag when processing an EDR.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Copy the CLI value to the MSISDN tag
false	Do not copy the CLI to the MSISDN

Default: false

Notes: If set to true and copyCLiToMsisdnRegExp is also set, then the CLI will not be copied to the MSISDN if a match is found for the expression defined in the `copyCliToMsisdnRegExp` parameter.

Example:
```
CopyCliToMsisdn = true
```

CopyCliToMsisdnRegExp

Syntax: `CopyCliToMsisdnRegExp = "(\\|exp\\|)"`

Description: Defines the expression to match. When a match occurs the `CopyCliToMsisdn` parameter is ignored and the EDR processing does not copy the CLI value to the MSISDN tag.

Type: String

Optionality: Optional.

Allowed: A valid regular expression. Double \\ (escapes) are required.

Default:

Notes: In the example below, the WIFI service will be matched for type 13 EDRs if the SERVICE tag appears in the middle or the end of the EDR. The CLI copy to MSISDN will not take place.

Example:
```
CopyCliToMsisdnRegExp =
"(\\|<CRD_TYPE=13>\\||\\|SERVICE=WIFI$\\|SERVICE=WIFI\\|)"
```

Failure

If the ccsCDRLoader fails, updates from the EDR files will not be completed. The EDR files will accumulate in the input directory.

Output

The ccsCDRLoader writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsCDRLoader.log**.

ccsCDRTrimDB

Purpose

The ccsCDRTrimDB process trims excess EDR records from the database. The excess records can be defined by one of the following:

- Wallet or subscriber ID
- The size of the cached records

This process modifies the CCS_BE_CDR table in the SMF. It gets the wallet/subscriber ID information from CCS_ACCT_ID. Rows are ordered by ID and RECORD_DATE.

The ccsCDRTrimDB process is not a daemon. It needs to be run manually or by cron.

Startup

The ccsCDRTrimDB process is run in the crontab for ccs_oper. By default it runs each night. It is scheduled by the **/IN/service_packages/CCS/bin/ccsCDRTrimDBStartup.sh** shell script.

Usage

```
ccsCDRTrimDB [-n int] [-c int]
[-h|--help]
```

Parameters

The ccsCDRTrimDB process supports the following command-line options.

`-c`

Syntax:	`-c` *int*
Description:	Sets the size of a buffer that will cache the records to be deleted. Records will be deleted when the: • Buffer is full • Last record in the table is reached
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	196
Notes:	

Example:	`-c 64`

`-n`

Syntax:	`-n` *int*
Description:	The maximum number of EDRs a subscriber can have.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	196
Notes:	
Example:	`-n 256`

`-h or --help`

Displays the help text file.

Example

This text shows an example of a command line startup for ccsCDRTrimDB.

```
ccsCDRTrimDB -n 256 -c 64
```

Note: This text may also be put in a startup shell script, such as **ccsCDRTrimDBStartup.sh**.

Failure

If the ccsCDRTrimDB process fails, records will accumulate in the SMF database.

Output

The ccsCDRTrimDB process writes error messages to the system messages file. It also writes additional output to the **/IN/service_packages/CCS/tmp/ccsCDRTrimDBStartup.sh.log** file.

ccsCDRTrimFiles

Purpose

The ccsCDRTrimFiles process deletes EDR files that have reached a nominated maximum age.

The ccsCDRTrimFiles process is not a daemon; it needs to be run manually or by cron.

Startup

This task is run in the crontab for ccs_oper. By default it runs each night. It is scheduled by the **/IN/service_packages/CCS/bin/ccsCDRTrimFilesStartup.sh** shell script:

Usage

```
ccsCDRTrimFiles [-d dir] [-a age] [-h| --help]
```

Parameters

The ccsCDRTrimFiles process supports the following command-line options.

`-a`

Syntax:	`-a age`
Description:	Maximum age allowed in days. Files older than this value will be removed.
Type:	
Optionality:	Optional (default used if not set).
Allowed:	
Default:	`1`
Notes:	
Example:	`-a 1`

`-d`

Syntax:	`-d dir`
Description:	Directory containing EDR files.
Type:	
Optionality:	Optional (default used if not set).
Allowed:	
Default:	`/logs/CDR/indexed`
Notes:	
Example:	`-d /logs/CDR/indexed`

`-h or --help`

Displays the help text file.

Output

The ccsCDRTrimFiles process writes error messages to the system messages file. It also writes additional output to the `/IN/service_packages/CCS/tmp/ccsCDRTrimFilesStartup.sh.log` file.

Failure

If the ccsCDRTrimFiles process fails, EDRs will collect in the indexed directory.

ccsChangeDaemon

Purpose

ccsChangeDaemon updates assignment of periodic charges to wallets. On the SMS ccsChangeDaemon handles periodic charge changes when a subscriber:

- Is associated with a new wallet
- Changes product type for a wallet

The daemon receives its tasks by reading CCS_PC_QUEUE table, which is hosted on the SMS and is replicated to the VWS.

Note: A ccsSLEEChangeDaemon also runs on the VWS. For more information, see *Purpose* (on page 223) for the ccsSLEEChangeDaemon.

Startup

On start-up, the daemon will check for the -r flag, if it does not find it, it will run in SMS mode.

On a non clustered SMS environment this task is started automatically by an entry in the inittab, through the **/IN/service_packages/CCS/bin/ccsChangeDaemonStartup.sh** shell script.

On a clustered SMS, startup is controlled by a failover resource group.

Configuration

ccsChangeDaemon supports parameters from the `ccsChangeDaemon` parameter group in the **eserv.config** file on the SMS. It contains parameters arranged in the structure shown in the example below.

```
ccsChangeDaemon = {

    PollPeriod = seconds
    suppressCcsPcQueueMessage = true | false
    throttle = int

    beClient = {
        clientName = "name"
        heartbeatPeriod = microsecs
        connectionRetryTime = seconds
        messageTimeoutSeconds = seconds

        billingEngines = [
            { id = int,
             primary = { ip="ip", port=port },
             secondary = { ip="ip", port=port }
            }
        ]
    }
}
```

eserv.config parameters

ccsChangeDaemon supports the following parameters from the `CCS` section of the **eserv.config** file on SMS.

`pollPeriod`

Syntax:	`pollPeriod = seconds`
Description:	Period in seconds between database reads.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60
Notes:	The CCS_PC_QUEUE table lists all outstanding work for the ccsChangeDaemon.
Example:	`pollPeriod = 60`

ptsUnsubscribeFromPCsForNonApplyPCs

Syntax: `ptsUnsubscribeFromPCsForNonApplyPCs = boolean`

Description: Controls if periodic charges (PCs) are unsubscribed when the account type is changed and the new account type is allowed the periodic charge, but it doesn't have **Apply to existing** set. When set to **true** (the default), and when the account type is changed for a wallet, all periodic charges for the service provider that aren't marked as **Apply to existing** and allowed for the new product type will be unsubscribed from.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

Default: true

Notes:

Example: `ptsUnsubscribeFromPCsForNonApplyPCs = true`

throttle

Syntax: `throttle = num`

Description: The maximum number of Voucher and Wallet Server updates per second.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

0	Disable throttling (no limit).
positive integer	Update limit.

Default: 1000

Notes:

Example: `throttle = 1000`

beClient

Syntax: `beClient = [{ config }]`

Description: The configuration for the connection to the beServer on the VWS.

Type: Parameter array

Optionality: Mandatory

Allowed:

Default:

Notes: This configuration is for the libBeClientIF library which ccsChangeDaemon uses to manage the connection.

For more information about this library, see *VWS Technical Guide*.

Example:

billingEngines

Syntax:

```
billingEngines = [
     { id = id
      primary = { ip="ip", port=port },
      secondary = { ip="ip", port=port }
     }
 ]
```

Description: Overrides connection details that beLocationPlugin obtains from the database.

For more information about the parameters included in the array, see *billingEngines* (on page 92) configuration for the ccsBeOrb process.

Type: Array.

Optionality: Optional.

Allowed:

Default:

Notes: Identifies the Voucher and Wallet Servers and assigns their Internet connection details.

Include this section to ensure that ccsChangeDaemon only connects to the local domain. If omitted, ccsChangeDaemon will connect to all VWS domains.

Example:

```
billingEngines = [
     { id = CHANGE_ME,
      primary = { ip="PRIMARY_BE_IP", port=1500 },
      secondary = { ip="SECONDARY_BE_IP", port=1500 }
     }
]
```

clientName

Syntax: `clientName = "name"`

Description: The unique client name of the process.

Type: String

Optionality: Mandatory

Allowed: Must be unique.

Default: "ccsChangeDaemon"

Notes: If more than one client connects with the same name the BE server will drop the other, therefore name must be unique.

Example: `clientName = "be1_ccsSLEEChangeDaemon"`

connectionRetryTime

Syntax: `connectionRetryTime = seconds`

Description: The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.

Type: Integer

Optionality: Required

Allowed:

Default: 5

Notes: This parameter is used by libBeClientIF.

Example: `connectionRetryTime = 2`

heartbeatPeriod

Syntax: `heartbeatPeriod = microsecs`

Description: The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.

Type: Integer

Optionality: Optional (Default used if not present)

Allowed:

0	Disable heartbeat detection.
positive integer	Heartbeat period.

Default:	30000000
Notes:	1 000 000 microseconds = 1 second.
Example:	`heartbeatPeriod = 30000000`

`messageTimeoutSeconds`

Syntax:	`messageTimeoutSeconds = seconds`
Description:	The time that the client process will wait for the server to respond to a request.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	1-604800 Number of seconds to wait. 0 Do not time out.
Default:	2
Notes:	After the specified number of seconds, the client process will generate an exception and discard the message associated with the request. This parameter is used by libBeClientIF.
Example:	`messageTimeoutSeconds = 2`

BE eserv.config parameters

The following parameters are available in the `BE` section of the **eserv.config**.

`amPrimary`

Syntax:	`amPrimary = true|false`
Description:	True if this is the primary VWS in the pair.
Type:	Boolean
Optionality:	Optional, default used if not set
Allowed:	
Default:	true
Notes:	
Example:	`amPrimary = false`

`beLocationPlugin`

Syntax:	`beLocationPlugin = "lib"`
Description:	The plug-in library that finds the Voucher and Wallet Server details of the Voucher and Wallet Servers to connect to.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	libGetccsBeLocation.so
Notes:	This library must be in the LD_LIBRARY_PATH.
Example:	`beLocationPlugin = "libGetccsBeLocation.so"`

`serverId`

Syntax:	`serverId = id`
Description:	The ID of the VWS pair.
Type:	Integer
Optionality:	
Allowed:	
Default:	1
Notes:	Set to 1 if this is not a VWS
Example:	`serverId = 11`

Failure

While ccsChangeDaemon is down, periodic charge assignment updates will not be executed on the local machine.

This table describes the recovery and failure files used by ccsChangeDaemon to attempt to recover after a failure.

File	Details
.failed	These files are written on both the SMS. They have the following naming convention: `.failed.ACSCustomerID.CCS_PC_QUEUE.ID` An entry is written to this file for each wallet update which initially fails. They contain a line for each failure: `SubscriberId|WalletId|PeriodicChargeBalanceTypeId|ProductId|ChangeType|ChangeAction|DomainId|NumberOfBalanceTypes|[|BalanceTypeId|BucketId|BucketValue[|...]]` Each time ccsChangeDaemon adds an entry to this file, it will also raise an Error level alarm. ccsChangeDaemon reads the entries in this file and attempts to reprocess them. Once all the entries in the file have been reprocessed, the ccsChangeDaemon deletes them.

failed	These files are written on the SMS. They have the following naming convention: `failed.ACSCustomerID.CCS_PC_QUEUE.ID`
	An entry is written to this file every time an entry in the .failed file is re-sent, and fails a second time. This file's first two lines are:
	`# Periodic Charge Change Daemon: failed updates`
	`# SubscriberId|WalletId|PeriodicChargeBalanceTypeId|ChangeType|ChangeAction|DomainId|NumberOfBalanceTypes|[|BalanceTypeId|BucketId|BucketValue[|...]]`
	Then there is an entry for each wallet update which fails a second time:
	`SubscriberId|WalletId|PeriodicChargeBalanceTypeId|ChangeType|ChangeAction|DomainId|NumberOfBalanceTypes|[|BalanceTypeId|BucketId|BucketValue[|...]]`
	Each time ccsChangeDaemon writes an entry to this file, it will raise an Error level alarm.
	Failure files are left for manual recovery.

Note: If an operation fails due to a "No Connection" error, ccsChangeDaemon will raise a LOGGED_WARNING and stop processing the row.

Output

ccsChangeDaemon writes recovery and failure logs to **/IN/service_packages/CCS/logs/ccsSLEEChangeDaemon/ccsPCChange/**.

If one of these files cannot be written to, the ccsChangeDaemon will exit with a critical error (for alarm details, see *CCS Alarms Reference Guide*).

ccsChangeDaemon writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsChange.log**.

ccsExpiryMessageLoader

Purpose

Sends short messages to subscribers to warn them that their wallet or balance will expire shortly. The list of subscribers is generated by ccsExpiryMessageGenerator on the VWSs and transferred to the SMS.

Startup

This task is run in the crontab for ccs_oper. By default it runs at 9 am each morning. It is scheduled directly through **/IN/service_packages/CCS/bin/ccsExpiryMessageLoader**.

Example

```
ExpiryMessages = {
    walletExpiryPeriod = 15
    numberOfWalletWarnings = 3
    balanceExpiryPeriod = 10
    numberOfBalanceWarnings = 3
    balanceTypes = [ 1, 2 ]
    onlyForLatestBucketExpiry = false
    oracleUsername = ""
    oraclePassword = ""
    generatorFilename = "ccsExpiryMessages"
    generatorFiledir = "/IN/service_packages/CCS/logs/expiryMessageWrite/"
    inputDirectory = "/IN/service_packages/CCS/logs/expiryMessageRead/"
    cmnPushFiles = [
        "-d", "/IN/service_packages/CCS/logs/expiryMessage/"
        "-r", "/IN/service_packages/CCS/logs/expiryMessage/"
        "-h", "SMF_HOST"
        "-p", "2027"
        "-F"
    ]
    pauseTime = 1
    batchSize = 2048
}
```

Note: This section is also used by ccsExpiryMessageGenerator.

Parameters

The ccsExpiryMessageLoader supports the following parameters from the `CCS` section of **eserv.config**.

balanceExpiryPeriod

Syntax:	`balanceExpiryPeriod = days`
Description:	Number of days before a Balance expires. Before the Balance expires, three expiry warning messages are sent, each at different times. The first message is sent balanceExpiryPeriod days before the wallet expires. The second and third messages are sent at two-thirds and one-third of balanceExpiryPeriod, respectively.
Type:	
Optionality:	
Allowed:	
Default:	10
Notes:	This parameter is optional. If it is omitted, no messages will be sent.
Example:	`balanceExpiryPeriod = 10`

balanceTypes = [ ]

Syntax:	`balanceTypes = [num]`
Description:	Specifies the balance types that should have expiry warning messages. When a new ACS customer is added, any balance types requiring expiry notifications should be added here.
Type:	Array
Optionality:	Optional
Allowed:	
Default:	No messages are sent
Notes:	Balance types are not split up for different ACS customers even though balance type identifiers belong to ACS customers.
Example:	`balanceTypes = [ 1, 2 ]`

batchSize

Syntax:	`batchSize = num`
Description:	The number of lines read from a file before a pause.
Type:	
Optionality:	Optional
Allowed:	
Default:	2048
Notes:	If it is not used: • Pauses will occur only between files • Throttling will not occur
Example:	`batchSize = 2048`

cmnPushFiles = [ ]

For the **eserv.config** on the VWS, use the cmnPushFiles configuration to transfer files to the SMS. There they will be ready for processing by ccsExpiryMessageLoader. Include the `-F` option to detect the file in use. See *cmnPushFiles* (on page 277) for all parameters.

Note: These directories must match those set by the `generatorFiledir` parameter.

generatorFiledir

Syntax:	`generatorFiledir = "dir"`
Description:	Directory for newly created expiry message files.
Type:	String
Optionality:	Optional (Default used if not specified)
Allowed:	
Default:	"/IN/service_packages/CCS/logs/expiryMessage/"
Notes:	This value required on both SMS and VWS machines. This value may be different on the two machines as long as cmnPushFiles has been configured to send and receive the appropriate directories.
Example:	`generatorFiledir = "/IN/service_packages/CCS/logs/expiryMessageWrite/"`

generatorFilename

Syntax:	`generatorFilename = "filename"`
Description:	Prefix for the file read by ccsExpiryMessageLoader.
Type:	
Optionality:	Optional.
Allowed:	
Default:	`"ccsExpiryMessages"`
Notes:	This parameter must be the same as that for the VWSs as the ccsExpiryMessageGenerator writes to this directory.
Example:	`generatorFilename = "ccsExpiryMessages"`

inputDirectory

Syntax:	`inputDirectory = "dir"`
Description:	Directory for newly created expiry message files.
Type:	String
Optionality:	Optional (Default used if not specified)
Allowed:	
Default:	"/IN/service_packages/CCS/logs/expiryMessage/"
Notes:	This value required on both SMS and VWS machines. This value may be different on the two machines as long as cmnPushFiles has been configured to send and receive the appropriate directories.
Example:	`inputDirectory = "/IN/service_packages/CCS/logs/expiryMessageRead/"`

numberOfBalanceWarnings

Syntax:	`numberOfBalanceWarnings = num`
Description:	The number of pending balance expiry messages to be sent. The messages will be equally spaced during the period set by the `walletExpiryPeriod` parameter.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	1,2, 3
Default:	3

Notes:
Example: `numberOfBalanceWarnings = 3`

numberOfWalletWarnings

Syntax: `numberOfWalletWarnings = num`
Description: The number of pending wallet expiry messages to be sent. The messages will be equally spaced during the period set by the `walletExpiryPeriod` parameter.
Type: Integer
Optionality: Optional (default used if not set).
Allowed: 1, 2, 3
Default: 3
Notes:
Example: `numberOfWalletWarnings = 3`

onlyForLatestBucketExpiry

Syntax: `onlyForLatestBucketExpiry = true|false`
Description: Whether to send expiry messages for all buckets that are going to expire or just the last bucket to expire.
Type: Boolean
Optionality: Optional (default used if not set).
Allowed:
- true - only send notifications for the latest bucket to expire for the configured balance types, or
- false - send notifications for all expiring buckets.

Default: false
Notes: Does not include buckets with no expiry date.
Example: `onlyForLatestBucketExpiry = true`

oraclePassword

Syntax: `oraclePassword = "password"`
Description: Oracle password.
Type:
Optionality:
Allowed:
Default: ""
Notes: Required on VWS.
Example:

oracleUsername

Syntax: `oracleUsername = "name"`
Description: Oracle user name
Type:
Optionality: Mandatory
Allowed:
Default: ""
Notes: Required on VWS.
Example:

`pauseTime`

Syntax:	`pauseTime = time`
Description:	The time separating the loading of individual files.
Type:	
Optionality:	
Allowed:	
Default:	1
Notes:	Optionally, if `batchSize` is also set, `pauseTime` defines the time between batches from an individual file.
Example:	`pauseTime = 1`

`walletExpiryPeriod`

Syntax:	`walletExpiryPeriod = days`
Description:	Number of days before the wallet expires. Before the wallet expires, three expiry warning messages are sent, each at different times. The first message is sent walletExpiryPeriod days before the wallet expires. The second and third messages are sent at two-thirds and one-third of walletExpiryPeriod, respectively.
Type:	
Optionality:	
Allowed:	
Default:	15
Notes:	This parameter is optional. If it is omitted, no messages will be sent.
Example:	`walletExpiryPeriod = 15`

Failure

If ccsExpiryMessageLoader fails, no notifications will be sent.

Output

The ccsExpiryMessageLoader writes error messages to the system messages file, and also writes additional output to the **/IN/service_packages/CCS/tmp/ccsExpiryMessageLoader.log** file.

ccsExternalProcedureDaemon

Purpose

ccsExternalProcedureDaemon is used to call CB10 C code from within a database trigger when adding a new ACS customer.

Startup - non clustered

In a non clustered environment this task is started automatically by entry cc11 in the inittab, by the **/IN/service_packages/CCS/bin/ccsExternalProcedureDaemon.sh** shell script.

Startup - clustered

In a clustered environment this task is started automatically by the Sun Plex manager and runs on one half of the cluster. It uses the CcsExternalProcedureDaemon failover resource to fail over to other nodes as required. The files required by the Sun Plex manager are located in the **/opt/ESERVCcsExternalProcedureDaemon** directory.

Location

The binary for the ccsExternalProcedureDaemon process is located at **/IN/service_packages/CCS/bin/ccsExternalProcedureDaemon** on the SMS.

Configuration

The ccsExternalProcedureDaemon does not require any specific configuration and it does not support any command line parameters.

Failure

If the ccsExternalProcedureDaemon fails then the CCS_CB10_CONFIG table will not be updated when you add an ACS customer.

Output

The ccsExternalProcedureDaemon writes error messages to the system messages file and writes additional output to **/IN/service_packages/CCS/tmp/ccsExternalProcedureDaemon.log**.

ccsLegacyPIN

Purpose

ccsLegacyPIN plug-in library is used by *ccsAccount* (on page 297) and the ccsVoucher_CCS3 voucher tool to encrypt the PINs using the DES authentication rule. For more information about authentication rules, see *Security libraries* (on page 23). ccsLegacyPIN library is not available for new voucher batches.

Note: The ccs3Encryption plug-in is a symbolic link to the *ccsLegacyPIN* (on page 142) plug-in, but in the ccs3Encryption mode it uses different parameters.

Startup

ccsLegacyPIN is used by ccsVoucher_CCS3 as necessary. No startup configuration is required for this library to be used.

Configuration

ccsLegacyPIN has no specific configuration. It does accept some parameters from ccsVoucher_CCS3 for voucher encryption which are configured in the CCS Voucher Management and Service Management screens.

ccsPeriodicCCRecharge

Purpose

Executes periodic credit card recharges on the SMS.

- Periodic credit card recharges are stored in the CCS_CC_RECHARGE_PENDING table in the SMF db.
- Can remove rows from the pending queue if the rows are:
  - No longer pending
  - Past configurable age limit

Start up

This task is run in the crontab for ccs_oper. By default it runs on the second day of each month. It is scheduled directly through **/IN/service_packages/CCS/bin/ccsPeriodicCCRecharge**.

Example

```
ccsPeriodicCCRecharge = {
    numRowsPerCommit = 100
    oracleUserAndPassword = "/"
    purgeOldEntriesAge = 0
    purgePendingRows = false
}
```

Parameters

ccsPeriodicCCRecharge supports the following parameters from the `CCS.ccsPeriodicCCRecharge` section of **eserv.config**.

`numRowsPerCommit`

Syntax:	`numRowsPerCommit = num`
Description:	Number of rows to insert before commit.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	100
Notes:	
Example:	`numRowsPerCommit = 500`

`oracleUserAndPassword`

Syntax:	`oracleUserAndPassword = "usr/pwd"`
Description:	Overrides userid and password for the Oracle SMF database connection set in *oracleUserAndPassword* (on page 52).
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"/"
Notes:	
Example:	

`purgeOldEntriesAge`

Syntax:	`purgeOldEntriesAge = days`
Description:	Number of days before a row will be removed from CCS_CC_RECHARGE_PENDING.

Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	0 (off)
Notes:	Entries with both pending and verified states will be removed.
Example:	`purgeOldEntriesAge = 14`

`purgePendingRows`

Syntax:	`purgePendingRows = true|false`
Description:	Whether or not to purge rows that are pending recharge from the CCS_CC_RECHARGE_PENDING table in SMF.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Purge rows that are pending recharge. false — Do not purge rows that are pending recharge.
Default:	false
Notes:	Only effective when *purgeOldEntriesAge* (on page 143) has a value > 0.
Example:	`purgePendingRows = true`

Failure

If ccsPeriodicCCRecharge fails, automatic credit card recharges will fail.

Note: Individual recharges through the PI will not be affected.

Output

The ccsPeriodicCCRecharge writes error messages to the system messages file. It also writes additional output to **/IN/service_packages/CCS/tmp/ccsPeriodicCCRecharge.log**.

ccsPeriodicCharge

Purpose

ccsPeriodicCharge applies periodic charges defined for wallets. The following types of periodic charge are supported:

- Credit
- Debit
- Voucher type recharge

ccsPeriodicCharge sends notifications to the subscriber informing them whether or not the charge was successful.

Note: This process only applies to periodic charges which were configured in CCS 3.1.4 or earlier.

Startup

ccsPeriodicCharge runs in either a solo mode or a parent and children mode.

The ccsPeriodicCharge solo process is run in the crontab for ccs_oper. By default it runs on an hourly basis. ccsPeriodicCharge is started automatically with the ccsPeriodicCharge command.

If the Daemon field is set to 2 or more in any product type, ccsPeriodicCharge will operate as a parent process, and will start a ccsPeriodicCharge child process for each id in the Daemon fields. The ccsPeriodicCharge parent process will remain active until all child processes have completed.

Note: If the service takes over an hour to run, it will examine all wallets and scheduling to ensure that the charges for the next hour are applied.

For more information about how product types assign periodic charges to ccsPeriodicCharge daemons, see *Subscriber Management - Product Types*, in *Charging Control Services User's Guide*.

Configuration - eserv.config

ccsPeriodicCharge is also configured by the `ccsPeriodicCharge` section of the **eserv.config** file. The structure of the ccsPeriodicCharge section is shown below.

```
ccsPeriodicCharge = {
    BatchSize = size
    OracleUserAndPassword = "usr/pwd"
    LockFile = "dir"
    profileTagCacheValidityPeriod = int
    BeQueueSize = int

    beLocationPlugin = "lib"
    oracleUserPass = "usr/pwd"
    clientName = "name"

    heartbeatPeriod = microsecs
    messageTimeoutSeconds = seconds
    maxOutstandingMessages = int
    reportPeriodSeconds = seconds
    connectionRetryTime = seconds

    plugins = [
        {
            config="confStr",
            library="lib",
            function="str"
        }
        [...]
    ]

    confStr = {
        plugin configuration
    }

    notEndActions = [
        {type="str", action="[ACK |NACK]"}
        [...]
    ]

    plugins configuration - see plugin-specific config
}
```

eserv.config parameters

ccsPeriodicCharge supports the following parameters from the ccsPeriodicCharge section of **eserv.config**.

BeQueueSize

Syntax: `BeQueueSize = num`

Description: The maximum number of VWS charging requests waiting for a response. If this limit is reached, no requests are sent until the number of outstanding requests drops below this number.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 500

Notes:

Example:
```
BeQueueSize = 250
```

clientName

Syntax: `clientName = "name"`

Description: The client name for the process.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: ccsPeriodicCharge

Notes: The server generates clientId from a hash of *name*.

This parameter is used by libBeClientIF. However, ccsAccount uses a different default.

Example:
```
clientName = "ccsPeriodicCharge"
```

connectionRetryTime

Syntax: `connectionRetryTime = seconds`

Description: The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.

Type: Integer

Optionality: Required

Allowed:

Default: 5

Notes: This parameter is used by libBeClientIF.

Example:
```
connectionRetryTime = 2
```

heartbeatPeriod

Syntax: `heartbeatPeriod = microsecs`

Description: The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.

Type: Integer

Optionality: Required

Allowed:

0	Disable heartbeat detection.
positive integer	Heartbeat period.

Default: 3000000

Notes: 1 000 000 microseconds = 1 second.

If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair.

This parameter is used by libBeClientIF.

Example: `heartbeatPeriod = 10000000`

LockFile

Syntax: `LockFile = "dir"`

Description: The location of the lock file used to prevent multiple instances of the ccsPeriodicCharge process.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: "/IN/service_packages/CCS/logs/.ccsPeriodicCharge"

Notes: If ccsPeriodicCharge is running in parent and child mode, only the parent process will use the lock file.

Example:
```
LockFile =
"/IN/service_packages/CCS/logs/.ccsPeriodicCharge"
```

maxOutstandingMessages

Syntax: `maxOutstandingMessages = num`

Description: The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.

Type: Integer

Optionality: Required

Allowed:

Default: If this parameter is not set, the maximum is unlimited.

Notes: If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls.

The messages are queued until the Voucher and Wallet Server has reduced its outstanding load.

This parameter is used by libBeClientIF.

Example: `maxOutstandingMessages = 100`

messageTimeoutSeconds

Syntax: `messageTimeoutSeconds = seconds`

Description: The time that the client process will wait for the server to respond to a request.

Type: Integer

Units: Seconds

Optionality: Required

Allowed:

1-604800	Number of seconds to wait.
0	Do not time out.

Default: 2

Notes: After the specified number of seconds, the client process will generate an exception and discard the message associated with the request.

This parameter is used by libBeClientIF.

Example:
```
messageTimeoutSeconds = 2
```

notEndActions

Syntax:
```
notEndActions = [
    {type="str", action="[ACK|NACK]"}
    [...]
]
```

Description: The `notEndActions` parameter array is used to define the messages associated with dialogs that should not have their dialog closes, because the dialog is closed by default. This facilitates failover.

Type: Parameter array.

Optionality: Required

Allowed:

Default:

Notes: If the incoming dialog for a call closes and the last response received was of the notEndActions type, the client process sends an ABRT message. The ABRT message allows the VWS to remove the reservation. An example of this situation would be where slee_acs has stopped working.

This parameter is used by libBeClientIF.

For more information about slee_acs, see *ACS Technical Guide*.

Example:
```
notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

OracleUserAndPassword

Syntax:
```
oracleUserAndPassword = "usr/pwd"
```

Description: The user and password combination ccsPeriodicCharge should use to log into the SMF database.

Type: String

Optionality: Optional

Allowed:

Default: "/"

Notes: Overrides CCS.oracleUserAndPassword. For more information about this parameter, see *oracleUserAndPassword* (on page 52).

Example:
```
oracleUserAndPassword = "/"
```

plugins

Syntax:
```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    ...
]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Mandatory

Allowed:

Default:

Notes: The voucherTypeRechargeOptions (VTR) plug-in needs the libclientBcast plug-in to function properly. It must be placed last in the plugins configuration list.

For more information about the libclientBcast plug-in, see *VWS Technical Guide*.

Example:
```
plugins = [
    {
        config="voucherTypeRechargeOptions",
        library="libccsClientPlugins.so",
        function="makeVoucherTypeRechargePlugin"
    }
    {
        config="",
        library="libclientBcast.so",
        function="makeBroadcastPlugin"
    }
]
```

config

Syntax: `config="name"`

Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example:
```
config="voucherRechargeOptions"
```

function

Syntax: `function="str"`

Description: The function the plug-in should perform.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `function="makeVoucherRechargePlugin"`

`library`

Syntax: `library="lib"`
Description: The filename of the plug-in library.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `library="libccsClientPlugins.so"`

`profileTagCacheValidityPeriod`

Syntax: `profileTagCacheValidityPeriod = seconds`
Description: Timeout value in seconds for the profile tag cache.
Type:3 Integer
Optionality: Optional
Allowed: Any positive decimal integer.
Default: 600
Notes:
Example: `profileTagCacheValidityPeriod = 800`

`reportPeriodSeconds`

Syntax: `reportPeriodSeconds = seconds`
Description: The number of seconds separating reports of failed messages.
Type: Integer
Units: Seconds
Optionality: Required
Allowed:
Default: 10
Notes: BeClient issues a failed message report:

- For timed-out messages
- For unrequested responses
- For new calls rejected because of congestion
- For messages with invalid Voucher and Wallet Server identifiers
- If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working

VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`.

This parameter is used by libBeClientIF.

Example: `reportPeriodSeconds = 10`

Command line parameters

ccsPeriodicCharge supports the following command line parameters.

```
ccsPeriodicCharge [-d] [-l log]
```

Note: These parameters can be set in a cronjob entry or startup script, or be set directly at the command line.

`-d`

Syntax:	`-d`
Description:	Display the configuration of ccsPeriodicCharge at start up.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	
Default:	Do not display configuration at startup.
Notes:	
Example:	

`-l`

Syntax:	`-l log`
Description:	The name of the file to log this child ccsPeriodicCharge daemon's debug output to.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	no default
Notes:	
Example:	`-l ccsPeriodicChargeDebug.log` This configuration will produce a log called **ccsPeriodicChargeDebug2.log** for a ccsPeriodicCharge daemon with an ID of 2.

Example

This text shows an example `ccsPeriodicCharge` **eserv.config** section.

```
ccsPeriodicCharge = {
    OracleUserAndPassword = "/"
    LockFile = "/IN/service_packages/CCS/logs/.ccsPeriodicCharge"
    clientName = "ccsPeriodicCharge"
    profileTagCacheValidityPeriod = 600
    BeQueueSize = 500

    plugins = [
        {
            # Voucher Type recharge plugin (VTR)
            config="voucherTypeRechargeOptions",
            library="libccsClientPlugins.so",
            function="makeVoucherTypeRechargePlugin"
        }
        {
            # Broadcast plugin needed by VTR
            config="",
            library="libclientBcast.so",
            function="makeBroadcastPlugin"
        }
    ]
```

```
    voucherTypeRechargeOptions = {
        srasActivatesPreuseAccount=false
        voucherTypeRechargeTriggers = ["VTR "]
    }

}
```

Failure

If ccsPeriodicCharge fails, the regular charges that are due will not be applied. However they will be applied retrospectively the next time ccsPeriodicCharge is run.

Output

The ccsPeriodicCharge writes error messages to the system messages file. It also writes additional output to **/IN/service_packages/CCS/tmp/ccsPeriodicCharge.log**.

ccsProfileDaemon

Purpose

ccsProfileDaemon performs the following:

- Processes profile change events
- Creates requests
- Sends requests to a third party ASP or customer care management platform

Profile change events are generated through changes to the subscriber's profile (ccs_acct_reference.PROFILE). For example, a profile change event is generated when a subscriber adds a new 'Friends and Family' number or subscribes to a voice mail service.

Startup - nonclustered

In a non-clustered environment, ccsProfileDaemon is started automatically by entry ccs8 in the inittab, through the **/IN/service_packages/CCS/bin/ccsProfileDaemonStartup.sh** shell script.

When CCS is installed, the startup inittab entry is added by the install process.

Disabling - ccsProfileDaemon

ccsProfileDaemon performs database cleanup of tables altered by subscriber profile creations and changes.

If you disable the ccsProfileDaemon task in your environment, you must also disable related triggers in your database to prevent your database from malfunctioning due to uncontrolled growth.

To disable the triggers used by ccsProfileDaemon in your database:

Step	Action
1	Log on the USMS node as the ccs_oper user.
2	Connect to the database as ccs_admin using SQLPlus.

3	Execute the following SQL statements:
	`SQL> alter trigger CCS_ACCT_REFERENCE_BUOPFER disable;`
	`SQL> alter trigger CCS_ACCT_REFERENCE_BIFER disable;`

4	Exit SQLPlus.

Example config section

```
ccsProfileDaemon = {
    PollInterval = 500
    LockFileName = "IN/service_packages/CCS/logs/.ccsProfileDaemon-lock"
    DisableConcurrencyLock = false
    AuditDirectory = "/IN/service_packages/CCS/logs/ccsProfileDaemon-logs"
    AuditFields = [1310806, 2829001, 2812014, 1310730]
    AuditFileName = "ccsProfileDaemon"
    AuditType = "IGNORE"
    CdrConcatenation = true
    MaxAgeSeconds = 60
    MaxSizeEntries = 100
    NotificationCacheAgeSeconds = 60
    AdditionalSpFields = [  ]
    PeriodicChargeTagCacheAge = 600
    SpFieldCacheAge = 600
    DateTimeFormat = "YYYY-MM-DDThh:mm:ss"
    allowLegacyServerConnect= true
    allowBugWorkArounds = true
    triggering = {
        DefaultOverrides = {
            CCSNamespace = "http://customer-smp/wsdls/ON/some.wsdl"
            Username = "username"
            Password = "password"
            OperationName = "NotificationRequest"
            ArbitraryParameters = "possible"
        }
        Operations = [
            {
                name = "CCSNotification"
                type = "OSD"
                overrides = {
                    CCSNamespace = "http://eng-prf-zone01-
                    z1/wsdls/ON/CCSNotifications.wsdl"
                    Username = ""
                    Password = ""
                    OperationName = "NotificationRequest"
                }
            }
        ]
        scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
    }
}
```

eserv.config parameters

ccsProfileDaemon supports the following parameters from the `ccsProfileDaemon` section of **eserv.config**.

`AdditionalSpFields`

Syntax:	`AdditionalSpFields = [`*`tagval1`*`,`*`tagval2`*`,,,,`*`tagvalN`*`]`
Description:	Allows additional profile tags to be added to the ccs_sp_field table array of integers.
Type:	Decimal integer for tagval x values
Optionality:	Optional
Allowed:	Any valid profile tag location values in decimal format.
Default:	Empty

Notes:

Example: `AdditionalSpFields = [100,120,140]`

allowBugWorkArounds

Syntax: `allowBugWorkArounds = true | false`

Description: Whether or not ccsProfileDaemon supports bug workarounds to cope with faulty SSL implementations on the ASP.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true – Bug workarounds are supported

false – Bug workarounds are not supported

Default: false

Notes: Set this parameter to true only if it is required for ccsProfileDaemon to make successful SSL connections to an ASP.

Example: `allowBugWorkArounds = true`

allowLegacyServerConnect

Syntax: `allowLegacyServerConnect = true | false`

Description: Whether or not ccsProfileDaemon allows connections to legacy servers that do not support secure renegotiation.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true – Allows connections to legacy servers that do not support secure renegotiation.

false – Prohibits connections to legacy servers that do not support secure renegotiation.

Default: false

Notes: Set this parameter to true only if it is required for ccsProfileDaemon to make successful SSL connections to an ASP.

Example: `allowLegacyServerConnect = true`

AuditDirectory

Syntax: `AuditDirectory = "dir"`

Description: Directory where we will write the audit logs.

Type: string

Optionality:

Allowed:

Default: "IN/service_packages/CCS/logs/ccsProfileDaemon-logs"

Notes:

Example:

AuditFields

Syntax: `AuditFields = [profile tag ID, ...]`

Description: An array of comma-separated profile tag IDs that identify subscriber profile fields that have changed and are being audited. Only values listed in the array will be audited.

The produced EDR has the following format:

```
USER=<value>|MSISDN=<value>ACS_CUST_ID=<value>|DATE=<value>|TERM
_IP_ADDR=<value>|CHANGED_TAGS=<value>
```

Where the format of CHANGED _TAGS is:

```
PROFILE_TAG_NAME=<value>:OLD_VALUE=<value>:NEW_VALUE=<value>
```

Example EDR:

```
USER=SU|MSISDN=321449000001|ACS_CUST_ID=11||DATE=20131218111933|
TERM_IP_ADDR=010167088183|CHANGED_TAGS=PROFILE_TAG_NAME=FD
Number:OLD_VALUE='123456789':NEW_VALUE='999888777666',
PROFILE_TAG_NAME=Acct Activation
Yearly:OLD_VALUE='20131202111824':NEW_VALUE='20131225111824',PRO
FILE_TAG_NAME=FF List,PROFILE_TAG_NAME=LO
Subscription:OLD_VALUE=:NEW_VALUE=True
```

See *CdrConcatenation* (on page 156) for additional information.

Type: Array of integers

Optionality: Optional (default used if not set)

Allowed: array of integers

Default: AuditFields = [ ] (disabled)

Notes: Only the profile tag name will be present for data that cannot be directly printed in an EDR, for example Prefix Tree content. You can view all EDR content through the subscriber screens.

Example: `AuditFields = [1310806, 2829001, 2812014]`

AuditFileName

Syntax: `AuditFileName = "name"`

Description: Base file name for the audit log – start and end times will be appended.

Type: string

Optionality:

Allowed:

Default: "ccsProfileDaemon"

Notes:

Example: `AuditFileName = "ccsProfileDaemon"`

AuditType

Syntax: `AuditType = "type"`

Description: Type of auditing.

Type: string

Optionality:

Allowed:

- "IGNORE" - regardless of response type, audit logs will not be generated
- "ERROR" - only create audit log for failure and error responses
- "ALL" - create audit log for all responses (successful, failure and error)

Default: "IGNORE"

Notes:

Example: `AuditType = "IGNORE"`

CdrConcatenation

Syntax: `CdrConcatenation = true | false`

Description: Specifies whether multiple changes to a profile should be concatenated into the same EDR. See *AuditFields* (on page 155) for additional information.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true, false

Default: false

Notes:

Example: `CdrConcatenation = false`

DateTimeFormat

Syntax: `DateTimeFormat = "dateformat"`

Description: Indicates the format for date and time variables that are sent in a DAP notification.

Type: String

Optionality: Optional (default used if not set)

Allowed: Only the following formats are accepted:

- YYYY-MM-DDThh:mm:ss
- YYYY-MM-DDThh:mm:ssZ
- -YYYY-MM-DDThh:mm:ss
- YYYYMMDDhhmmss

Default: YYYYMMDDhhmmss

Notes:

Example: `DateTimeFormat = "YYYY-MM-DDThh:mm:ss"`

DisableConcurrencyLock

Syntax: `DisableConcurrencyLock = true|false`

Description: Whether to disable concurrency locking.

Type:

Optionality:

Allowed: true, false

Default: false

Notes:

Example: `DisableConcurrencyLock = false`

LockFileName

Syntax: `LockFileName = "file"`

Description: The lock file name to determine if we have multiple profile daemon processes running on the same SMS node.

Type: string

Optionality:

Allowed:

Default: "IN/service_packages/CCS/logs/.ccsProfileDaemon-lock"

Notes:
Example:

MaxAgeSeconds

Syntax: `MaxAgeSeconds = seconds`
Description: Maximum age, in seconds, after which all audit entries will be written to disk.
Type: integer
Optionality:
Allowed:
Default: 60
Notes:
Example: `MaxAgeSeconds = 60`

MaxSizeEntries

Syntax: `MaxSizeEntries = size`
Description: Maximum size (number) after which all audit entries will be written to disk.
Type: Integer
Optionality:
Allowed:
Default: 100
Notes:
Example: `MaxSizeEntries = 100`

NotificationCacheAgeSeconds

Syntax: `NotificationCacheAgeSeconds = seconds`
Description: Maximum age, in seconds, before the notification definitions cache will be reread from the database.
Type: Integer
Optionality:
Allowed:
Default: 60
Notes:
Example: `NotificationCacheAgeSeconds = 60`

PeriodicChargeTagCacheAge

Syntax: `PeriodicChargeTagCacheAge = seconds`
Description: Timeout value, in seconds, for data in the periodic charge tag cache.
Type: Integer
Optionality: Optional
Allowed: Any positive decimal integer value.
Default: 600 (seconds)
Notes:
Example: `PeriodicChargeTagCacheAge = 600`

PollInterval

Syntax:	`PollInterval = milliseconds`
Description:	How long, in milliseconds, that we should sleep before processing profile change events.
Type:	integer
Optionality:	
Allowed:	
Default:	500
Notes:	
Example:	`PollInterval = 500`

SpFieldCacheAge

Syntax:	`SpFieldCacheAge = seconds`
Description:	Timeout value in seconds for data in the SpField tag cache.
Type:	Decimal integer
Optionality:	Optional
Allowed:	Any positive decimal integer.
Default:	600
Notes:	
Example:	`SpFieldCacheAge = 600`

triggering

Syntax:	`triggering = {parameter_list}`
Description:	The configuration of the individual XmlTcap or OSD operations that can be recieved.
Type:	List
Optionality:	Mandatory
Allowed:	
Default:	For operations that are not configured, these Operations > overrides defaults are applied: • CCSNamespace="http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl" • name = "CCSNotification", type = "OSD" • Username = "" • Password = ""
Notes:	

DefaultOverrides

Syntax:	`DefaultOverrides = {global_parameter_list}`
Description:	The list of global default parameter values for each of the **overrides** parameters in the individual trigger **Operations** configured.
Type:	List
Optionality:	Mandatory.

Allowed: Must be all of these:
- CCSNamespace
- Username
- Password
- OperationName
- ArbitraryParameters

Default:

Notes: These parameters are inserted into the **Operations > overrides** section when the parameter is omitted from the **overrides** list.

Example:
```
DefaultOverrides = {
    CCSNamespace = "http://customer-smp/wsdls/ON/some.wsdl"
    Username = "username"
    Password = "password"
    OperationName = "NotificationRequest"
    ArbitraryParameters = "possible"
}
```

ArbitaryParameters

Syntax: `ArbitraryParameters = "value"`

Description:

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: None

Notes:

Example: `ArbitraryParameters = "possible"`

CCSNamespace

Syntax: `CCSNamespace = "namespace"`

Description: The name space used for the WSDL request.

Type: String

Optionality: Mandatory

Allowed:

Default:

Notes:

Example: `CCSNamespace = "http://customer-smp/wsdls/ON/some.wsdl"`

OperationName

Syntax: `OperationName = "name"`

Description: The name of the OSD request.

Type: String

Optionality: Mandatory

Allowed:

Default:

Notes:

Example: `OperationName = "NotificationRequest"`

Password

Syntax: `Password = "password"`

Description: The HTTP password to use.

Type: String

Optionality: Mandatory.

Allowed:

Default:

Notes:

Example: `Password = "password"`

Username

Syntax: `Username = "name"`

Description: The HTTP user name to use.

Type: String

Optionality: Mandatory

Allowed:

Default:

Notes:

Example: `Username = "username"`

Operations

Syntax: `Operations = [op1],[op2]`

Description: Maps of individual operations for the trigger.

Type: Array

Optionality: Mandatory.

Allowed:

Default:

Notes:

Example:

```
Operations = [
    {
        name = "CCSNotification"
        type = "OSD"
        overrides = {
            OperationName = "NotificationRequest"
        }
    }
]
```

name

Syntax: `name = "operation_name"`

Description: The name of the operation as received from the VWARS.

Type: String

Optionality Mandatory

Allowed:

Default:

Notes:

Example `name = "CCSNotification"`

overrides

Syntax: `overrides = {override_list}`

Description: Set of override parameters for this operation that are added to/override the values received from the beVWARS.

Type: List

Optionality: Optional (default used if not set).

Allowed:

Default: The values that are defined by DefaultOverrides for any missing parameter.

Notes: If all of an operation's overrides parameters values are the same as the DefaultOverrides, it is not necessary to specify this parameter.

For OSD, it is expected to set:

- CCSNamespace : the namespace used for the wsdl request.
- Username : the HTTP username to use
- Password : the HTTP password to use
- OperationName : The name of the OSD request, that is, "NotificationRequest"

For XmlTcap, it is expected to set:

- Control_Plan: The control plan to trigger
- Service_Handle: The service handle to use to do the triggering

Example:

```
overrides = {
    Username = ""
    Password = ""
    OperationName = "NotificationRequest"
}
```

type

Syntax: `type = "protocol"`

Description: The protocol for the operation.

Type: String

Optionality: Mandatory

Allowed:

- OSD
- XmlTcap

Default:

Notes:

Example: `type = "OSD"`

scps

Syntax: `scps = [SLC1,SLC2]`

Description: An array of xmlTcapInterface SLCs, in the format:

HOSTNAME:*PORT*

Type: Array

Optionality: PORT is optional (default used if not set).

Allowed: A list of existing SLC host names and ports

Default: 3072

Notes:

Example: `scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]`

`osd_scps`

Syntax: `osd_scps = [SLC1,SLC2>]`

Description: An array of OSD SLCs, in the format:
HOSTNAME:*PORT*

Type: Array

Optionality: PORT is optional (default used if not set).

Allowed: A list of existing SLC host names and ports

Default: 3072

Notes:

Example:
```
osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
```

`osd_scps_custid`

Syntax:
```
osd_scps_custid = [{ host = ["SLC1:port","SLC2:port" ]
                                serviceProvider = customer-id
                    }
                   ]
```

Description: An array of combinations of OSD SLCs and service provider in the following format:

host = HOSTNAME:PORT

serviceProvider = ACS customer ID in the database.

If osd_scps_custid is configured, then beServiceTrigger will use the configured SLC and service-provider to match the acs-customer from the plugin request and send request to proper SCP.

Type: Array

Optionality: Optional, If there is no osd_scps_custid section, then osd_scps section will be used.

Allowed: A list of existing SLC host names and ports along with service provider.

Default:

Notes: To process old events, at-least one entry is mandatory in the above configuration without service provider and that host will be used to process old events.

Example:
```
osd_scps_custid= [
                    { host = ["SLC4:2211","SLC5:3211" ]
                      serviceProvider = 21
                    }
                    { host = ["SLC5:1024"]
                    }
                  ]
```

Command line parameters

The ccsProfileDaemon accepts the following command line parameters.

Usage:

`ccsProfileDaemon` [`-i` | `--node_id` *node_id*] [`-n` | `--number` *number*]

Example:
```
ccsProfileDaemon -i 2 -n 2
```

`-i or --node_id`

Syntax:	`-i | --node_id` *node_id*
Description:	The SMS node id that this ccsProfileDaemon instance is running on.
Type:	integer
Optionality:	
Allowed:	Cannot be greater than the number of nodes specified and must be greater than 0.
Default:	1
Notes:	This value will be used in conjunction with the number of nodes specified to limit the range of subscriber's that are processed by a specific ccsProfileDaemon.
Example:	`-i 2`

`-n or --number`

Syntax:	`-n` *int* `--number` *int*
Description:	The number of ccsProfileDaemon instances running across all SMS nodes.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	Must be greater than '0'.
Default:	1
Notes:	This value will be used in conjunction with the node id specified to limit the range of subscriber's that are processed by a specific ccsProfileDaemon.
Examples:	`-n 2` `--number 2`

ccsReports

Purpose

The ccsReports section specifies the parameters for CCS SMS reports.

Note: Reports use `CCS.oracleUserAndPassword` as the Oracle login.

Example

Here is an example of the `ccsReports` section in the **eserv.config** file.

```
ccsReports = {
    accountLogDir = ""
    accountPrefixName = ""
    cdrDir = ""
    cdrPrefix = ""
    voucherLogDir = ""
    voucherPrefixName = "pre"

    VoucherStatus = {
        outputDirectory =
        "/IN/service_packages/SMS/output/Ccs_Service/Summary/VoucherStatus"
```

```
        archiveDirectory =
        "/IN/service_packages/SMS/output/Ccs_Service/Summary/VoucherStatus/archive"
        archiveAfterDays = 10
        deleteAfterDays = 60
    }
}
```

Parameters

ccsReports accepts the following parameters.

accountLogDir

Syntax: `accountLogDir = "dir"`
Description: The account log directory.
Type: String
Optionality:
Allowed:
Default: None
Notes:
Example:

accountPrefixName

Syntax: `accountPrefixName = "name"`
Description: The account prefix name.
Type: String
Optionality:
Allowed:
Default: None
Notes:
Example:

cdrDir

Syntax: `cdrDir = "dir"`
Description: The EDR directory.
Type: String
Optionality:
Allowed:
Default: None
Notes:
Example:

cdrPrefix

Syntax: `cdrPrefix = "pre"`
Description: The EDR prefix.
Type: String
Optionality:
Allowed:
Default: None

Notes:

Example:

voucherLogDir

Syntax: `voucherLogDir = "dir"`

Description: The voucher log directory.

Type: String

Optionality:

Allowed:

Default: None

Notes:

Example: `voucherLogDir = "/var/logs/voucher"`

voucherPrefixName

Syntax: `voucherPrefixName = "pre"`

Description: The voucher prefix name.

Type: String

Optionality:

Allowed:

Default: None

Notes:

Example: `voucherPrefixName = "voucher_"`

VoucherStatus

Syntax:

```
VoucherStatus = {
    outputDirectory = "dir"
    archiveDirectory = "dir"
    archiveAfterDays = days
    deleteAfterDays = days
 }
```

Description: Configuration for voucher status reports.

Type: Parameter group

Optionality:

Allowed:

Default:

Notes: Additional configuration for the Voucher Status Report is available in the **voucherStatusReport.env** file. For more information about this file, see *Voucher Status Report Configuration* (on page 73).

Example:

archiveAfterDays

Syntax: `archiveAfterDays = days`

Description: How old reports should be before being archived

Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 10
Notes:
Example: `archiveAfterDays = 10`

archiveDirectory

Syntax: `archiveDirectory = "dir"`
Description: Where archived reports are moved to.
Type: String
Optionality: Optional (default used if not set)
Allowed:
Default: "/IN/service_packages/SMS/output/Ccs_Service/Summary/VoucherStatus/archive"
Notes:
Example:
```
archiveDirectory =
"/IN/service_packages/SMS/output/Ccs_Service/Summary/Voucher
Status/archive"
```

deleteAfterDays

Syntax: `deleteAfterDays = days`
Description: How many days old reports can be before they are removed by the system.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 60
Notes:
Example: `deleteAfterDays = 60`

outputDirectory

Syntax: `outputDirectory = "dir"`
Description: The location of the voucher status reports.
Type: String
Optionality: Optional (default used if not set)
Allowed:
Default: "/IN/service_packages/SMS/output/Ccs_Service/Summary/VoucherStatus"
Notes:
Example:
```
outputDirectory =
"/IN/service_packages/SMS/output/Ccs_Service/Summary/Voucher
Status"
```

ccsWalletExpiry

Purpose

ccsWalletExpiry processes CCS updates to the SMF database from the VWSs. There are two types of update.

- Expiry requests cause wallets to be set to Terminated in the SMF database.
- Removal requests cause wallets to be removed from the SMF database.

If ccsWalletExpiry removes all the wallets associated with a subscriber account and will also remove the subscriber account.

Startup

This task is run in the crontab for ccs_oper. By default it runs every 10 minutes. It is scheduled directly through **/IN/service_packages/CCS/bin/ccsWalletExpiry**.

Example

An example of a configuration for the ccsWalletExpiry process and cssVWARSExpiry plug-in follows.

```
ccsVWARSExpiry = {
    expiredPrefix = "expiredWallet"
    expiredDirectory = "/IN/service_packages/CCS/logs/wallet"
    removedDirectory = "/IN/service_packages/CCS/logs/wallet"
    removedPrefix = "removedWallet"

    expiredMsisdnPath="/IN/service_packages/CCS/logs/MSISDNExpiry"
    expiredMsisdnPrefix="MSISDNExpiry"
    expiredMsisdnMaxAge = 120
    cmnPushFiles = [
        "-d", "/IN/service_packages/CCS/logs/wallet"
        "-r", "/IN/service_packages/CCS/logs/wallet"
        "-h", "produsms01"
        "-p", "2027"
        "-F"
    ]
}
```

Note: This configuration section is also used by *ccsVWARSExpiry* (on page 244) on the VWS.

Parameters

ccsWalletExpiry supports the following parameters from the `CCS` section of **eserv.config**.

`cmnPushFiles = [ ]`

Syntax:	`cmnPushFiles = []`
Description:	For the **eserv.config** on the VWS, use the cmnPushFiles configuration to transfer files to the SMS ready for processing by ccsExpiryMessageLoader.
Type:	Parameter array
Optionality:	Mandatory
Allowed:	
Default:	

Notes: Include the `-F` option to detect the file in use. See *cmnPushFiles* (on page 277) for all parameters.

These directories must match the respective directories set in `generatorFiledir`.

Example:

`expiredDirectory`

Syntax: `expiredDirectory = "dir"`

Description: Defines the location of files listing wallets moving to terminated state.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "/IN/service_packages/CCS/logs/wallet"

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

Example: `expiredDirectory = "/var/CCS/expiredWallets"`

`expiredMsisdnMaxAge`

Syntax: `expiredMsisdnMaxAge = seconds`

Description: The maximum age of export file in seconds.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 120

Notes:

Example: `expiredMsisdnMaxAge = 180`

`expiredMsisdnPath`

Syntax: `expiredMsisdnPath = "dir"`

Description: Location for the output file on the SMS for sending to the HLR. The output file is written by the ccsWalletExpiry (cronjob).

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "/IN/service_packages/CCS/tmp"

Notes:

Example: `expiredMsisdnPath = "/var/CCS/expiredMsisdns"`

`expiredMsisdnPrefix`

Syntax: `expiredMsisdnPrefix = "pre"`

Description: Prefix of output file.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "MSISDNExpiry"

Notes: The filename format is: *`expiredMsisdnPrefix`*`YYYYMMDDHHMMSS.export`.

Example:	`expiredMsisdnPrefix = "prodube01_msisdnsExp"`

`expiredPrefix`

Syntax:	`expiredPrefix = "`*`prefix`*`"`
Description:	The prefix of files listing wallets moving to terminated state.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"expiredWallet"
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS. The filename format is: *expiredPrefix_YYYYMMDDHHMMSSexpiredSuffix*
Example:	`expiredPrefix = "prodube01_termWallets"`

`removedDirectory`

Syntax:	`removedDirectory = "`*`dir`*`"`
Description:	Defines the location of files listing wallets being removed.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"/IN/service_packages/CCS/logs/wallet"
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.
Example:	`removedDirectory = "/var/CCS/removedWallets"`

`removedPrefix`

Syntax:	`removedPrefix = "`*`prefix`*`"`
Description:	The prefix of files listing wallets being removed from the system.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"removedWallet"
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS. Whether ccsVWARSExpiry or ccsWalletExpiry removes the wallet depends on *logNotRemoveWallet* (on page 248). The filename format is: *removedPrefix_YYYYMMDDHHMMSSremovedSuffix*
Example:	`removedPrefix = "prodube01_removeWallets"`

Failure

If ccsWalletExpiry fails, wallet expiry updates from the VWS will fail.

Output

The ccsWalletExpiry writes error messages to the system messages file. It also writes additional output to **/IN/service_packages/CCS/tmp/ccsWalletExpiry.log**.

libccsCommon

Purpose

libccsCommon provides common functions to various CCS processes.

Startup

libccsCommon is used by a number of CCS processes. No startup configuration is required for this library to be used.

Configuration

The libccsCommon library supports parameters from the common parameter group in the **eserv.config** file on all machines. For more information, see Configuration.

VoucherRedeemFail Files

Purpose

The VoucherRedeemFail files are used as an aid to fraud detection by providing a list of all redeem failures for post processing by a third party.

All type 15 ("Voucher Redeem") EDRs with a result of anything other than "success" cause a record to be written to the current fail file.

Before being added to, each fail file is archived when the maxEDRs number has been reached, or the file has been open longer than the maxOpenDuration time and there is at least one record in the file

Record format

The pipe separated file format is follows:

VoucherNumber|*MSISDN*|*RedemptionDate*|*FailureReason*

The field are taken from the type 15 EDR record as follows:

Field	EDR Tag
VoucherNumber	VOUCHER_NUMBER
MSISDN	MSISDN when MSISDN plug-in is active, otherwise REDEEMING_ACCT_REF
RedemptionDate	RECORD_DATE
FailureReason	RESULT

Note: If any information is not available, the corresponding column will be left blank.

File name format

The naming convention for the current/temporary file is:

tmp_failed_Voucher_*PID_file-open-time*.log

The naming convention of the current/temporary file when it is archived for third party processing is:

`failed_Voucher_`*`PID_datetime`*`.log'`

Chapter 4

Background Processes on the SLC

Overview

Introduction

This chapter provides a description of the programs or executables used by CCS as background processes on the SLCs.

Executables are located in the `/IN/service_packages/CCS/bin` directory.

Some executables have accompanying scripts that run the executables after performing certain cleanup functions. All scripts should be located in the same directory as the executable.

For more information about the processes and systems that use these programs and executables, see *System Overview* (on page 1).

Warning: It is a pre-requisite for managing these core service functions that the operator is familiar with the basics of Unix process scheduling and management. Specifically, the following Unix commands:

- init (and inittab)
- cron (and crontab)
- ps
- kill
- top
- vi (or other editing tool)

In this chapter

This chapter contains the following topics.

BeClient 173
PIClientIF 179
ccsActions 180
ccsCB10HRNAES 184
ccsCB10HRNSHA 184
ccsMacroNodes 184
ccsSvcLibrary 196
libccsClientPlugins 205
libccsCommon 207

BeClient

Purpose

The BeClient is a SLEE interface which handles connections to the beServer process on the VWS for SLEE applications running on the SLC.

The BeClient needs to be configured for CCS so functions such as voucher recharge can be completed. This is implemented as a CCS specific plug-in that is described further below.

For more information about the BeClient, see *VWS Technical Guide*.

BeClient plugins

The BeClient can be extended by installing plug-ins. This section lists the available BeClient plug-ins which are provided with CCS.

Voucher plugin

This plug-in controls the voucher recharge process. It splits the voucher recharge wallet message into three messages:

- Voucher reserve
- Wallet recharge
- Voucher commit

If the voucher reserve or wallet recharge operation fails, the whole process stops. This allows for the possibility that the vouchers and wallets are on different VWS pairs and provides for an automatic voucher redeem process that does not require post-process reversals.

This function is contained within the **libccsClientPlugins.so** library.

Note: For this plug-in to function properly, the Broadcast plug-in (**libclientBcast.so**) must also be installed and configured. For more information about the Broadcast plug-in, see *VWS Technical Guide*.

Merge wallets plug-in

The plug-in manages the merging of two wallets. It:

- Receives merge wallets requests.
- Obtains identifiers for the wallets involved.
- Determines whether a link or a merge is required.
- If a link is required, the plug-in relinks the wallets and subscribers.
- If a merge is required, the plug-in:
  - Locks the source wallet for 30 seconds,
  - Merges the source and destination wallets
  - Relinks the wallets and subscribers.

The merge wallets function is contained in the **libccsClientPlugins.so** library.

Location

This binary is located on SLCs.

Startup

The BeClient is a SLEE interface and is started during SLEE initialization. The line in the **SLEE.cfg** which starts the BeClient is:

```
INTERFACE=ccsBeClient BeClientStartup.sh /IN/service_packages/CCS/bin/ 1 EVENT
```

Note: The above settings are defaults and may vary.

For instructions about starting and stopping BeClient, see *SLEE Technical Guide*.

Configuration

In order to load and operate, BeClient plug-ins read the `BeClient` section of the **eserv.config** file. The BeClient section is listed below, showing the configuration for the plug-ins provided with CCS.

Note: This text does not show the full configuration for BeClient. For more information about the full configuration for the BeClient, see *VWS Technical Guide*.

```
BeClient = {
    standard BeClient configuration

    plugins = [
        {
            # Voucher recharge plugin (VRW)
            config="voucherRechargeOptions",
            library="libccsClientPlugins.sl",
            function="makeVoucherRechargePlugin"
        }
        {
            # Broadcast plugin needed by VRW
            config="broadcastOptions",
            library="libclientBcast.so",
            function="makeBroadcastPlugin"
        }
        {
            # Voucher Type recharge plugin (VTR)
            config="voucherTypeRechargeOptions",
            library="libccsClientPlugins.so",
            function="makeVoucherTypeRechargePlugin"
        }
    ]

    voucherRechargeOptions = {
        voucherRechargeTriggers = [
            "VRW "
        ]
        srasActivatesPreuseAccount = false
        srActivatesPreuseAccount = true
        sendBadPin = false
    }

    broadcastOptions = {
        aggregateNAckCodes = [
            "NVOU"
        ]
    }

    voucherTypeRechargeOptions = {
        srasActivatesPreuseAccount=false
        voucherTypeRechargeTriggers = ["VTR "]
    }
}
```

Parameters

BeClient has no command line parameters.

The BeClient supports the following parameters from the `BE` section of **eserv.config**.

plugins

Syntax:

```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    [...]
]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Mandatory

Allowed:

Default:

Notes: The VRW needs the libclientBcast plug-in to function properly. It must be placed last in the `plugins` configuration list.

For more information about the libclientBcast plug-in, see *VWS Technical Guide*.

Example:

```
plugins = [
    {
        # Voucher recharge plugin (VRW)
        config="voucherRechargeOptions",
        library="libccsClientPlugins.so",
        function="makeVoucherRechargePlugin" }
    {
        # Broadcast plugin needed by VRW
        config="broadcastOptions",
        library="libclientBcast.so",
        function="makeBroadcastPlugin" }
    {
        # Voucher Type recharge plugin (VTR)
        config="voucherTypeRechargeOptions",
        library="libccsClientPlugins.so",
        function="makeVoucherTypeRechargePlugin"}
]
```

config

Syntax: `config="name"`

Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example:

```
config="voucherRechargeOptions"
```

function

Syntax: `function="str"`

Description: The function the plug-in should perform.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:
Default: No default
Notes:
Example: `function="makeVoucherRechargePlugin"`

library

Syntax: `library="lib"`
Description: The filename of the plug-in library.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `library="libccsClientPlugins.so"`

broadcastOptions

Syntax:
```
broadcastOptions = {
    aggregateNAckCodes = [config]
}
```
Description: Name of configuration section for the BeClient Broadcast plug-in libclientBcast.
Type: Parameter array
Optionality:
Allowed:
Default:
Notes: libclientBcast is used by a range of processes which connect to the beServer, including:

- BeClient
- PIbeClient
- ccsBeOrb

For more information about libclientBcast, see `libclientBcast`.

Example:
```
broadcastOptions = {
    aggregateNAckCodes = [ ]
}
```

aggregateNAckCodes

Syntax:
```
aggregateNAckCodes = [
    "NVOU"
]
```
Description: When this parameter is set, the BeClient waits for a response from all the VWS pairs in use and filters the responses from the broadcast request using the configured NAck codes.
Type: Parameter array
Optionality:
Allowed: NVOU
Default:

Notes: When a voucher recharge request is broadcast, this ensures that all the available VWS pairs are checked for the required voucher before a voucher not found message is returned to the requesting process.

Example:

Example

```
BeClient = {
    clientName = "scpClient1"
    heartbeatPeriod = 3000000
    maxOutstandingMessages = 100
    connectionRetryTime = 5

    plugins = [
        {
            # Voucher recharge plugin (VRW)
            config="voucherRechargeOptions",
            library="libccsClientPlugins.sl",
            function="makeVoucherRechargePlugin"}
        {
            # Broadcast plugin needed by VRW
            config="broadcastOptions",
            library="libclientBcast.so",
            function="makeBroadcastPlugin" }
        {
            # Voucher Type recharge plugin (VTR)
            config="voucherTypeRechargeOptions",
            library="libccsClientPlugins.so",
            function="makeVoucherTypeRechargePlugin"}
    ]

    voucherRechargeOptions = {
        voucherRechargeTriggers = [
            "VRW "
        ]
        srasActivatesPreuseAccount = false
        srActivatesPreuseAccount = true

        sendBadPin = false
    }

    broadcastOptions = {
        aggregateNAckCodes = [
            "NVOU"
        ]
    }

    voucherTypeRechargeOptions = {
        srasActivatesPreuseAccount=false
        voucherTypeRechargeTriggers = ["VTR "]
    }

    notEndActions = [ {type="IR  ", action="ACK "}
                    {type="SR  ", action="ACK "}
                    {type="SR  ", action="NACK"}
                    {type="INER", action="ACK "}
                    {type="SNER", action="ACK "}
                    {type="SNER", action="NACK"}
    ]
}
```

Output

The BeClient writes error messages to the system messages file.

PIClientIF

Purpose

The PI Client Interface is a TCP client interface that runs on the SLC and communicates with one or more PIProcesses running on the SMS.

PI commands are received from the Service Logic Execution Environment (SLEE) and forwarded to any available PI process with the appropriate synstamp. PI responses are sent back to the SLEE on the appropriate dialog.

Add subscriber (CCSCD1=ADD) and delete subscriber (CCSCD1=DEL) commands are both supported. Add subscriber can be configured to override or add additional parameters to those provided in the incoming message, but delete subscriber does not require additional configuration, which is why there is no CCSCD1.DEL section in the example configuration file.

Startup

The PIClientIF is a SLEE interface and is started during SLEE initialization. The line in the **SLEE.cfg** which starts the PIClientIF is:

```
INTERFACE=ccsPiClientIF ccsPiClientIF.sh /IN/service_packages/CCS/bin 1 EVENT
```

Note: The above settings are defaults and may vary.

For instructions about starting and stopping PIClientIF, see *SLEE Technical Guide*.

Output

The PIClientIF writes error messages to the system messages file.

Example

```
piClientIF = {

 # Hostname or IP address of the PI
 # (Mandatory)
 # Default: None
 # host = "usms"

 # Ports on which the PI processes are listening
 # (Mandatory)
 # Default: None
 ports = [ 2999 , 3000 ]

 # Username for the PI login
 # (Optional)
 # Default: "admin"
 username = "admin"

 # Password for the PI login
 # (Optional)
 # Default: "admin"
 password = "admin"
```

```
 # Time (in milliseconds) to wait for a response
 # from the PI.
 # (Optional)
 # Default: 2000
 timeoutInterval = 2000

 # Maximum number of timed-out PI requests on
 # a connection before that connection is closed
 # where "0" indicates that the connection will
 # be immediately closed after a timeout occurs.
 # It is recommended that maxTimeouts is set to
 # the default (0), the connection will be
 # re-opened after connectionRetryTimeout has
 # expired.
 # (Optional)
 # Default: 0
 maxTimeouts = 0

 # Time (in milliseconds) to wait before re-trying
 # a failed connection.
 # (Optional)
 # Default: 1000
 connectionRetryTimeout = 1000

 # Maximum number of outstanding PI requests.
 # New requests will be rejected if this limit
 # is reached.
 # (Optional)
 # Default: 50
 maxQueueSize = 50

 # Period (in milliseconds) to log statistics for
 # debugging. Information such as pending request
 # queue length, number of available connections
 # and notifications of dialog closed events will
 # be logged. A value of "0" indicates this feature
 # is disabled. Note: some PI requests (eg CCSCD1=DEL)
 # will not be sent on more than 1 connection at a time.
 # (Optional)
# Default: 0
statsTimer = 0

 }
```

ccsActions

Purpose

ccsActions provides the functions which enable the CCS Feature Nodes to interact with other elements in the system, including:

- acsChassis
- the VWS (via the BeClient and beServer), and
- other elements on the network (such as the VPU).

Startup

If ccsActions is included in the **acs.conf**, ccsActions will be started by slee_acs when the SLEE is started.

For more information about how this included in **acs.conf**, see *ChassisPlugin* (on page 53).

Configuration

In order to load and operate, ccsActions reads the ccsActions section of the **eserv.config** file. The ccsActions section is listed below.

```
ccsActions = {
    maxOutstandingBeClientMsgs = int
    loggedNotificationPeriod = int
    loggedInvalidPeriod = int
    exceptionLogPeriod = int
    configuredVolumeITC = int
    volumeReservationLength = int
    accumulateChargeInfoCosts = true|false
    allowNegativeNofundsPolicyOnCommit = true|false
}
```

Parameters

ccsActions supports the following parameters from the `CCS` section of **eserv.config**.

accumulateChargeInfoCosts

Syntax:	`accumulateChargeInfoCosts = true|false`
Description:	Determines if charge costs are allowed to accumulate.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true - will allow to accumulate • false - will reset the cost for each charge
Default:	true
Notes:	
Example:	`accumulateChargeInfoCosts = true`

allowNegativeNoFundsPolicyOnCommit

Syntax:	`allowNegativeNoFundsPolicyOnCommit = true|false`
Description:	If funds have been depleted during a reservation request, sets whether or not to allow the overall balance to go negative when committing the reservation.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true (allow the overall balance to go negative) false (do not allow the overall balance to go negative)
Default:	false
Notes:	
Example:	`allowNegativeNoFundsPolicyOnCommit = true`

configuredVolumeITC

Syntax:	`configuredVolumeITC = val`
Description:	Sets the bearer capability - Information Transfer Capability value for a data charging session.
Type:	
Optionality:	
Allowed:	Valid values (in decimal) are:

- 0 = speech
- 8 = UDI (unrestricted digital information)
- 9 = RDI (restricted digital information)
- 16 = 3.1 kHz Audio
- 17 = UDI with tones / announcements (Q.931 1998)
- 24 = Video

Default: 8
Notes:
Example: `configuredVolumeITC = 8`

loggedInvalidPeriod

Syntax: `loggedInvalidPeriod = seconds`
Description: Interval separating the writing of each summary of ccsActions errors to the syslog.
Type:
Optionality: Optional
Allowed:
Default: 10
Notes:
Example: `loggedInvalidPeriod = 10`

loggedNotificationPeriod

Syntax: `loggedNotificationPeriod = int`
Description: The logged notification period.
Type: Integer
Optionality:
Allowed:
Default:
Notes:
Example: `loggedNotificationPeriod = 10`

maxOutstandingBeClientMsgs

Syntax: `maxOutstandingBeClientMsgs = number`
Description: The maximum number of outstanding BeClient messages.
Type: Integer
Optionality: Optional (default used if missing)
Allowed:
Default: 1000
Notes: Too small a value may result in calls being dropped.
Example: `maxOutstandingBeClientMsgs = 2000`

`quotaProfileBlock`

Syntax:	`quotaProfileBlock = int`
Description:	The profile block number to use for retrieving quota related profile fields. Five pairs of quota value and quota opt-out tags are looked up in this block: • Quota value tags (numbers 0x140511 to 0x140515) • Quota opt-out tags (numbers 0x140521 to 0x140525)
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	An integer in the range 8 to 15 for the APP1 to APP8 profile blocks, or 16 for the ANY_VALID profile block.
Default:	8 – This is the subscriber profile block
Notes:	
Example:	`quotaProfileBlock = 16`

`volumeReservationLength`

Syntax:	`volumeReservationLength = days`
Description:	The reservation length, in days, for data charging sessions such as Radius Control Agent.
Type:	
Optionality:	
Allowed:	
Default:	5
Notes:	The UBE parameter `noExpectedKeep` should be set to the same value (in seconds) as this parameter. See *VWS Technical Guide*.
Example:	`volumeReservationLength = 5`

Example

```
ccsActions = {
    maxOutstandingBeClientMsgs = 1000
    loggedNotificationPeriod = 10
    loggedInvalidPeriod = 10
    configuredVolumeITC = 8
    volumeReservationLength = 5
}
```

Failure

If ccsActions fails, the CCS feature node functionality will fail. This will usually result in call processing becoming unstable or failing.

Output

ccsActions writes summaries of its error messages to the system messages file.

ccsCB10HRNAES

License

The ccsCB10HRNAES library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsCB10HRNSHA

License

The ccsCB10HRNSHA library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsMacroNodes

Purpose

The CCS service library handles initial call setup for calls that use CCS functionality and configures any necessary profile data used in CCS feature node parameters. For information about the available CCS feature nodes, see *Feature Nodes Reference Guide*.

Startup

If ccsMacroNodes is configured in **acs.conf**, it is made available to slee_acs when slee_acs is initialized. It is included in the `acsChassis` section of **acs.conf** in a `MacroNodePluginFile` entry as follows:

```
acsChassis
 MacroNodePluginFile ccsMacroNodes.so
```

Configuration

ccsMacroNodes accepts the following parameters.

Example ccsMacroNodes config

Here is an example of the `CCS.ccsMacroNodes` section of the **eserv.config** file.

```
ccsMacroNodes = {
    expireAtMidnightTZ="GMT"
    MaximumMenuRetries = 2
    MaximumBadCodeRetries = 3
    MaxCreditCardNumberLength = 20
    MinCreditCardNumberLength = 20
    PromptAndCollectMaxAnnouncements = 10
    PromptAndCollectInterMenuBlockTimeout = 1
    ATBNoAnswerTimeout = 10
    AVPProfileTagData = ["Idle", "Blocked"]
    CCACustomAvpProfileTag = 512
    resetUATBOnINSF = false
    PAVRBalancesUseSystemCurrency = true
    NoChargeEventClass = "FnF FnD Events"
    NoChargeEventName = "FnF Config Change"
    FFDiscountRule = "EXPLICIT"
    HomeCountryNationalPrefix = ""
```

```
    UseDisconnectLeg = false
    BFTGracePeriodLength = 0
    BSPlayAllExpiriesAtEnd = false
    BSAnnBalanceTypes = [
        {
        acsCustomerId = 1
        balTypeIds = [6, 7, 10]
        }
    ]
    DOCCRAnnBalanceTypes = [
        {
        acsCustomerId = 1
        balTypeIds = [6, 7]
        }
    ]
    VRRedeemMinVoucherLength=9
    VRRedeemMaxVoucherLength=15
    VRRedeemAcctFrozenCheck=true
    SMSCIIncludeZeroBalances = true
    SMSCIExcludeZeroBalanceTypes = [78,79]
    SMSABUseFormattedExpiryDate=true
    SMSABExpiryFormat = "%d/%m/%y"

    # UATB Node:
    # If vws returns IR_Nack with INSF -
    #   [1] create a zero-value reservation
    #   [2] enable SR_Ack grants of 0
    #   [3] grant 0 deciseconds in AC
    # Optional.
    holdReservationOpen = false

    # UATB Node
    # Enable UATB macronode loopback
    # Optional.
    macronodeLoopbackBranch1 = false  #IR_Ack no funds
    macronodeLoopbackBranch15 = false #SR_Nack no funds
    macronodeLoopbackBranch16 = false #SR_Ack with funds

    # UATB Node
    # Reroute IR_Nack failures to alternative exits
    # Optional.
    IR_Nack = {
    # The following are default settings:
    PROC = 2 # Route IR_Nack[PROC] (UnknownWallet)      to exit 2: BFT
    INSF = 1 # Route IR_Nack[INSF] (InsufficientFunds)  to exit 1: Declined(No Funds)
    TMNY = 1 # Route IR_Nack[TMNY] (MaxConcurrent)      to exit 1: Declined(No Funds)
    CRIS = 1 # Route IR_Nack[CRIS] (CallRestricted)     to exit 1: Declined(No Funds)
    COM = 2 # Route IR_Nack[COM ] (CommunicationError) to exit 2: BFT
    NACK = 2 # Route IR_Nack[NACK] (SystemError)        to exit 2: BFT
    WDIS = 1 # Route IR_Nack[WDIS] (WalletDisabled)     to exit 1: Declined(No Funds)

}
```

ATBNoAnswerTimeout

Syntax:

Description:

Type:

Optionality:

Allowed:

Default: 10

Notes: This parameter is not used.

Example:

HomeCountryNationalPrefix

Syntax: `HomeCountryNationalPrefix = "prefix"`

Description: Defines the prefix for the home country.

Type: String

Optionality:

Allowed:

Default: ""

Notes:

Example: `HomeCountryNationalPrefix = ""`

AVPProfileTagData

Syntax: `AVPProfileTagData = ["value1", "value2",…]`

Description: This parameter holds the values which should be checked with CCACustomAvpProfileTag.

Type: Array

Optionality:

Allowed:

Default: None

Notes:

Example: `AVPProfileTagData = ["Idle", "Blocked"]`

CCACustomAvpProfileTag

Syntax: `CCACustomAvpProfileTag = num`

Description: This parameter decide the profile tag which is holding the AVP value. The value 0 indicates that the feature is disabled.

Type: Integer

Optionality:

Allowed: Numerical value

Default: 0

Notes:

Example: `CCACustomAvpProfileTag = 512`

resetUATBOnINSF

Syntax: `resetUATBOnINSF = true|false`

Description: This flag decides whether to reset the UATB state in case of insufficient balance and exit from UATB exit1.

Type: Boolean

Optionality:

Allowed: true, false

Default: false

Notes:

Example: `resetUATBOnINSF = false`

MaxCreditCardNumberLength

Syntax: `MaxCreditCardNumberLength = len`
Description: Defines the maximum length allowed for credit card numbers.
Type: Integer
Optionality:
Allowed:
Default: 20
Notes: Applies to the Credit Card Starter Menu node only.
Example: `MaxCreditCardNumberLength = 20`

MaximumBadCodeRetries

Syntax:
Description:
Type: integer
Optionality:
Allowed:
Default:
Notes: This parameter is not used currently.
Example:

MaximumMenuRetries

Syntax: `MaximumMenuRetries = num`
Description: Defines the maximum number of times the subscriber can attempt to enter voucher numbers, PINs, and other menu options correctly, before they are blacklisted.
Type: Integer
Optionality:
Allowed:
Default: 2
Notes: Applies to all nodes which limit subscriber retry attempts.
Example: `MaximumMenuRetries = 2`

MinCreditCardNumberLength

Syntax: `MinCreditCardNumberLength = len`
Description: Defines the minimum length allowed for credit card numbers.
Type: Integer
Optionality:
Allowed:
Default: 20
Notes: Applies to the Credit Card Starter Menu node only.
Example: `MinCreditCardNumberLength = 20`

`PromptAndCollectInterMenuBlockTimeout`

Syntax:	`PromptAndCollectInterMenuBlockTimeout = seconds`
Description:	Defines the timeout in seconds, after playing all the announcements for the current menu.
Type:	Integer
Optionality:	
Allowed:	
Default:	1
Notes:	Applies to the Account Type Swap, Dynamic Menu, and Credit Card Recharge nodes.
Example:	`PromptAndCollectInterMenuBlockTimeout = 1`

`PromptAndCollectMaxAnnouncements`

Syntax:	`PromptAndCollectMaxAnnouncements = num`
Description:	Defines the maximum number of announcements to play at one time.
Type:	Integer
Optionality:	
Allowed:	
Default:	10
Notes:	Applies to the Account Type Swap and Dynamic Menu nodes only.
Example:	`PromptAndCollectMaxAnnouncements = 10`

Node specific parameters

Additional node-specific parameters follow.

Balance Status

`BSAnnBalanceTypes = [{}{}]`

The list of balance types to be announced in the node. This parameter is mandatory.

`acsCustomerId`

Default:	1

`balTypeIds`

[n,n,n]

`BSPlayAllExpiriesAtEnd`

Syntax:	`BSPlayAllExpiriesAtEnd = true|false`
Description:	Determines if each expiry is played after its corresponding balance announcement.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true - Play all expiry limits after all balance announcements are played. • false - Play each expiry to be after its corresponding balance announcement.
Default:	
Notes:	
Example:	`BSPlayAllExpiriesAtEnd = false`

`RetryReserveOnNoFunds`

Syntax:	`RetryReserveOnNoFunds = true|false`
Description:	When true, the UATB node will try a second reservation attempt when: • Only the duration withheld from the IRR remains • We have received a NAck from the BE on our final reservation This is intended for use with configurations where a low credit notification may be triggered by the reservation attempt, which recharges the account or frees other funds. The second attempt may then succeed.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true, false
Default:	false
Notes:	
Example:	`RetryReserveOnNoFunds = false`

Balance Status Branch

`BSBCheckBalance`

The list of balance types to check for each customer. The balance types must all have the same balance unit. For more information, see *Introduction* (on page 69). This parameter is optional.

`acsCustomerId`

Default:	1

`balTypeIds`

[n,n,n]

`expireAtMidnightTZ`

Syntax:	`expireAtMidnightTZ = "tz"`
Description:	Sets wallets and buckets to expire at midnight for the time zone specified.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	The time zone part of the parameter must be typed in a form that the operating system recognizes. Alternatively you can select a time zone from the operating system's list. To view top-level time zone names, enter `ls /usr/share/lib/zoneinfo` from a shell. To see second-level time zone names enter `ls /usr/share/lib/zoneinfo TopLevelName/`. For example, to verify that the operating system recognizes a time zone name for DeNoranha, in Brazil, you would enter `ls /usr/share/lib/zoneinfo/Brazil/`. DeNoranha is listed, so the time zone name would be `"Brazil/DeNoranha"`.
Default:	false (do not modify expiry calculation).
Notes:	A list of time zones can be found in the Time Zones appendix of *ACS Technical Guide*.

Example: An account is created at 2 p.m. on 5 September 2006 and is set to have a life span of 24 days.

If the parameter `expireAtMidnightTZ = "Asia/Vladivostok"` is included, the account will expire on 29 September 2006 at midnight, Vladivostok time.

If this parameter is omitted, the account will expire on 29 September 2006 at 2 p.m.

Call Info

`SMSCIIncludeZeroBalances`

Syntax: `SMSCIIncludeZeroBalances = true|false`

Description: Controls the inclusion of zero balances in the final notification composed by the Call Information SMS feature node.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Include zero balances.
false	Exclude zero balances.

Default: false

Notes: This value determines the behavior of all instances of the Call Information SMS feature node. For information about the Call Information SMS feature node, see *Feature Nodes Reference Guide*.

Example: `SMSCIIncludeZeroBalances = true`

`SMSCIExcludeZeroBalanceTypes`

Syntax: `SMSCIExcludeZeroBalanceTypes = [n,n,n]`

where, n = ccs_balance_type.ID

Description: Controls the exclusion of balance types having value zero, in the final notification composed by the Call Information SMS (SMSCI) feature node.

When `SMSCIIncludeZeroBalances` is set as true, then by default all the balance types having value as zero, are included in the notification composed by SMSCI node. In this case, `SMSCIExcludeZeroBalanceTypes` will help in excluding the unwanted balance types from the notification string.

Type:

Optionality: Optional (default used if not set)

Allowed: ccs_balance_type.ID

Default:

Notes:

Example: `SMSCIExcludeZeroBalanceTypes = [78,79]`

Do Credit Card Recharge

`DOCCRAnnBalanceTypes = [{}{}]`

The list of balance types (Cash only) to be announced in the node (mandatory).

`acsCustomerId`

Default: 1

`balTypeIds`

[n,n,n]

Friends and Family config

`FFDiscountRule`

Syntax:	`FFDiscountRule = "rule"`
Description:	Determines how discount is applied for an individual call.
Type:	string
Optionality:	
Allowed:	Valid values are: • EXPLICIT = the discount is applied as configured • DIVIDED = the discount applied is divided by the number of F+F members configured for the subscriber.
Default:	"EXPLICIT"
Notes:	
Example:	`FFDiscountRule = "EXPLICIT"`

`NoChargeEventClass`

Syntax:	`NoChargeEventClass = "class"`
Description:	The event class to use when sending named event requests to the Voucher and Wallet Server.
Type:	string
Optionality:	
Allowed:	A valid event class
Default:	"FnF FnD Events"
Notes:	
Example:	`NoChargeEventClass = "FnF FnD Events"`

`NoChargeEventName`

Syntax:	`NoChargeEventName = "name"`
Description:	The event name to use when sending named event requests to the Voucher and Wallet Server.
Type:	string
Optionality:	
Allowed:	A valid event name
Default:	"FnF Config Change"
Notes:	
Example:	`NoChargeEventName = "FnF Config Change"`

Play Voucher Redeemed Info config

`PAVRBalancesUseSystemCurrency`

Syntax:	`PAVRBalancesUseSystemCurrency = true|false`
Description:	Whether to force the use of the system currency for the Play Voucher Redeemed Info feature node.
Type:	Boolean
Optionality:	Optional (default used if not set).

Allowed:	true	Use the system currency.
	false	Use the currency of the active wallet.
Default:	false	
Notes:	In addition, the configuration item `systemCurrencyIdAgeSeconds` (on page 291) may be used to control the cache time applied to system currency ID. For more information about the Play Voucher Redeemed Info feature node, see *Feature Nodes Reference Guide*.	
Example:	`PAVRBalancesUseSystemCurrency = false`	

SMS Account Balance

`SMSABExpiryFormat`

Syntax:	`SMSABExpiryFormat = "format"`
Description:	If *SMSABUseFormattedExpiryDate* (on page 192) is set to true, use this format.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	Maximum format length is 49 characters
Default:	"%d/%m/%y"
Notes:	
Example:	`SMSABExpiryFormat = "%d/%m/%y"`

`SMSABIncludeZeroBalances`

Syntax:	`SMSABIncludeZeroBalances = true|false`
Description:	Whether to include zero balances when using SMS Account Balance node.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Include zero balances in the notification. false — Do not include zero balances in the notification.
Default:	false
Notes:	
Example:	`SMSABIncludeZeroBalances = true`

`SMSABUseFormattedExpiryDate`

Syntax:	`SMSABUseFormattedExpiryDate = true|false`
Description:	Whether or not to format the expiry date.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Use *SMSABExpiryFormat* (on page 192) to define how the expiry date is formatted. false — Do not alter the format of the expiry date.
Default:	false
Notes:	If set to true, the date variable should be included in the Balance Expiry Template (for example, using "It will expire on %s."). For more information about the Balance Expiry Template, see *Charging Control Services User's Guide*.
Example:	`SMSABUseFormattedExpiryDate = false`

UATB

The following parameters are used for the UATB node.

Note: The UATB node may also require switch configuration. See *Switch Configuration for the UATB Node* (on page 71).

`BFTGracePeriodLength`

Syntax:	`BFTGracePeriodLength = seconds`
Description:	How to handle grace periods for reservations under BFT.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	• -1 – No grace period on BFT (communication or system error) for subsequent reservations. Node will branch disconnected (NSF) on communication/system error. • 0 – No grace period on BFT (communication or system error) for subsequent reservations. Node will properly treat call as BFT, branching disconnected (BFT) on communication/system error. Call length of 0 is confirmed. • Positive – The call is allowed to continue for the specified number of seconds on communication/system error for subsequent reservations. Node will properly treat call as BFT, branching disconnected (BFT) on communication or system error. Call length of O is confirmed.
Default:	-1
Notes:	BFT is usually triggered when a Voucher and Wallet Server fails. Used with UATB node.
Example:	`BFTGracePeriodLength = 30`

`continueIfAnnouncementFails`

Syntax:	`continueIfAnnouncementFails = true|false`
Description:	If the UATB feature node fails to play the pre-announcement and this flag is set to: • true – The UATB feature node continues to try to charge the subscriber. • false – The UATB feature node follows the appropriate failure branch.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true or false
Default:	false
Example:	`continueIfAnnouncementFails = true`

`MinResRemainingBeforeSubReservation`

Syntax:	`MinResRemainingBeforeSubReservation = num`
Description:	The value the UATB uses to decide if it should issue a subsequent reservation (SR) request to the VWS. An SR request is made if the remaining reservation is greater than this parameter (read notes below).
Type:	Integer
Optionality:	Optional (default value used if not set)

Allowed: -1 for no limit (that is, infinite), or any integer.

Default: 300

Notes:

- When configured, if the time elapsed since the last SR was sent exceeds the the "Requested Reservation Chunk" value on the SMS screens, then UATB will send an SR, regardless of any value set for this parameter.
- The units will be in the units applicable for the service being processed. For example for Camel voice, the units will be in deci-seconds. If the SLC processes calls or sessions for more than one type of service or protocol, then service-specific configuration will be required for each service (see `Service`).
- This feature can be used to prevent the SLC from generating too many reservation request messages to the VWS if the remaining reservation is below the configured threshold.
- For Used Units Confirmation (UUC) functionality, configure this value to a large value or -1 so an SR Request message will always be sent to the VWS and reservations size can be controlled by the "Requested Reservation Chunk" value on the SMS screens.
- The value configured at the ccsMacroNodes level will be the default or global value used if no service-specific configuration exists (see `Service`).

Example: `MinResRemainingBeforeSubReservation = 300`

`Service`

Syntax: `Service = [{`*`service1`*`}{`*`service2`*`}{`*`servicen`*`}]`

Description: Different MinResRemainingBeforeSubReservation values can be configured for different services on the SLC. Each array element or sub-section in the Service Array specifies the ACS service name and corresponding MinResRemainingBeforeSubReservation value for that service.

Type: Array

Optionality: Optional (default used if not set).

Notes: If no service array exists or if no service-specific entry exists in the Service array section for the specific service, the ccsUATB node will use the global value described in the parent section.

Example: Here is example array:

```
Service = [
    {
        ServiceName = "CCS_DATA"
        MinResRemainingBeforeSubReservation = -1
    }
    {
        ServiceName = "CCS"
        MinResRemainingBeforeSubReservation = 300
    }
    {
        ServiceName = "CCS_OTHER"
        MinResRemainingBeforeSubReservation = 30
    }
]
```

`UseDisconnectLeg`

Syntax: `UseDisconnectLeg = `*`true`*`|`*`false`*

Description: How to end BFT call.

Type: Boolean

Optionality:	Optional (default used if not set).
Allowed:	• true – Sends a TCAP Disconnect (2). • false – Sends a TCAP release.
Default:	false
Notes:	
Example:	`UseDisconnectLeg = true`

Voice Call Cost

`VCCTimeAnnParts`

Syntax:	`VCCTimeAnnParts = num`
Description:	Defines the number of variable parts to use for time balance announcements.
Type:	
Optionality:	Optional.
Allowed:	
Default:	2
Notes:	
Example:	`VCCTimeAnnParts = 2`

Voucher Recharge

`VRRedeemAcctFrozenCheck`

Syntax:	`VRRedeemAcctFrozenCheck = true|false`
Description:	Whether or not ccsMacroNodes should check whether the subscriber's account state is frozen following voucher redeem failure.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true – Use a WI request to check the subscriber's account state. • false – Do not send a wallet information request.
Default:	true
Notes:	
Example:	`VRRedeemAcctFrozenCheck = true`

`VRRedeemDefaultScenario`

Syntax:	`VRRedeemDefaultScenario = true|false`
Description:	Indicates if the voucher recharge node should attempt to use a default scenario.
Type:	Boolean
Optionality:	Optional
Allowed:	true, false
Default:	false
Notes:	Needs to be set to true for VWS vouchers using default scenarios. For Voucher Manager vouchers this parameter has no effect.
Example:	`VRRedeemDefaultScenario = true`

VRRedeemMaxVoucherLength

Syntax:	VRRedeemMaxVoucherLength = *len*
Description:	The maximum number of digits in a voucher number.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	Must be equal to or larger than *VRRedeemMinVoucherLength* (on page 196).
Default:	14
Notes:	See also VRRedeemMaxVoucherLength.
Example:	VRRedeemMaxVoucherLength = 15

VRRedeemMinVoucherLength

Syntax:	VRRedeemMinVoucherLength = *len*
Description:	The minimum number of digits in a voucher number.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	Must be equal to or smaller than *VRRedeemMaxVoucherLength* (on page 196).
Default:	14
Notes:	See also VRRedeemMinVoucherLength.
Example:	VRRedeemMinVoucherLength = 9

ccsSvcLibrary

Purpose

Based on the incoming call details, the ccsSvcLibrary loads up the relevant control plan and feature nodes.

Startup

If ccsSvcLibrary is configured in **acs.conf**, it is made available to slee_acs when slee_acs is initialized. It is included in the acsChassis section of **acs.conf** in a ServiceEntry.

```
acsChassis
 ServiceEntry (CCS,ccsSvcLibrary.so)
```

Configuration

ccsSvcLibrary supports parameters from the ccsServiceLibrary parameter group in the **eserv.config** file on a SLC. It contains parameters arranged in the structure shown below.

```
ccsServiceLibrary = {
    UnknownDataReleaseCause = int
    callPlanAndDataCacheValidityTime = seconds
    callPlanAndDataCacheFlushTime = seconds
    callPlanAndDataCacheMaxAge = seconds
    enableProfile6 = true|false
    AccountLength = int
    IncomingCallBarEnable = "int"
    IncomingCallBarDisable = "int"
    MobileTerminatingHomeCli = "cli"
    ContinueAsConnect = true|false
    InterpretAccountNumberAsCLI = true|false
    NoCallPlanError = "sev"
```

```
    GlobalDefaultForAcctRefCallPlanName = "name"
    GlobalDefaultForSMOrigCallPlanName = "name"
    GlobalDefaultForSMTermCallPlanName = "name"
    globalCapabilityFlushPeriod = 10
    promptForAccountOnOriginatingSK = true|false
    promptForAccountOnTerminatingSK = true|false

    productCapabilitiesCacheFlushTime = seconds
    productCapabilitiesCacheMaxAge = seconds
    productCapabilitiesCacheValidityTime = seconds

    productTypeForExternalSub = "pt_name"

    SubscriberDomainType = id
    VoucherDomainType = id
    PreCallAnnouncementId = id
    WithheldDuration = int
    SingleReservation = true|false
    PreCallLowBalance = true|false
    RetrieveLCRNumbers = true|false
    ConvergedScenario = true|false
}
```

AccountLength

Syntax:	`AccountLength = int`
Description:	Defines the length of the subscriber number, and is used when splitting the subscriber number entered from the PIN.
Type:	Integer
Optionality:	
Allowed:	
Default:	10
Notes:	
Example:	`AccountLength = 10`

callPlanAndDataCacheFlushTime

Syntax:	`callPlanAndDataCacheFlushTime = seconds`
Description:	How often a check is made for data older than its validity time.
Type:	Integer
Optionality:	Optional (default used if missing)
Allowed:	Any positive integer
Default:	3600
Notes:	Applies to control plans matched on originator or destination addresses only. To reload the cache more frequently with the latest versions of control plans, set the *callPlanAndDataCacheFlushTime* to a low value. For example, when set to 60, the cache is flushed every 60 seconds.
Example:	`callPlanAndDataCacheFlushTime = 300`

callPlanAndDataCacheMaxAge

Syntax:	`callPlanAndDataCacheMaxAge = seconds`
Description:	The time after which an unused or unchanged control plan is dropped from the control plan cache.
Type:	Integer

Optionality:	Optional (default used if missing)
Allowed:	Any positive integer
Default:	3600
Notes:	Applies to control plans matched on originator or destination addresses only. To reload the cache more frequently with the latest versions of control plans, set the *callPlanAndDataCacheMaxAge* to a low value. For example, when set to 60, the cache is flushed every 60 seconds.
Example:	`callPlanAndDataCacheMaxAge = 300`

callPlanAndDataCacheValidityTime

Syntax:	`callPlanAndDataCacheValidityTime = seconds`
Description:	The maximum age of the data before it is refreshed from the database.
Type:	Integer
Optionality:	Optional (default used if missing)
Allowed:	Any positive integer
Default:	3600000
Notes:	Applies to control plans matched on originator or destination addresses only.
Example:	`callPlanAndDataCacheValidityTime = 300`

ContinueAsConnect

Syntax:	`ContinueAsConnect = true|false`
Description:	If this is a TCAP-CONTINUE, then replace the TCAP-CONTINUE with a TCAP-CONNECT and send it to a switch.
Type:	Boolean
Optionality:	
Allowed:	true, false
Default:	false
Notes:	
Example:	`ContinueAsConnect = false`

ConvergedScenario

Syntax:	`ConvergedScenario = true|false`
Description:	This parameter is set to true in convergent charging deployments. When it is set to true, it disables the functionality that is not required in converged billing scenarios. For example, database subscriber lookup is not performed as converged deployments do not have subscribers in the database.
Type:	Boolean
Optionality:	
Allowed:	true, false
Default:	false
Notes:	
Example:	`ConvergedScenario = false`

enableProfile6

Syntax:	`enableProfile6 = true|false`
Description:	Enable application profile block 6 for use with alternate subscriber data.

Type: Boolean
Optionality: Optional (default used if not set).
Allowed: true, false
Default: false
Notes: Warnings will be output in the log file when voucher recharge calls are processed if this is false.
Example: `enableProfile6 = true`

getCallPlanNumberFromProfile

Syntax: `getCallPlanNameFromProfile = true|false`
Description: Controls whether call plan name should be fetched from the subscriber's profile.
Type: Boolean
Optionality: Optional (default used if not set).
Allowed:
- true - from subscriber's profile
- false - use normal control plan selection rules.

Default: false
Notes:
Example: `getCallPlanNameFromProfile = true`

GlobalDefaultForAcctRefCallPlanName

Syntax: `GlobalDefaultForAcctRefCallPlanName = "name"`
Description: This specifies the global default control plan for the account reference.
Type: string
Optionality:
Allowed:
Default: "E2 Global Prompt For Account Reference"
Notes:
Example: `GlobalDefaultForAcctRefCallPlanName = "E2 Global Prompt For Account Reference"`

GlobalDefaultSMOrigCallPlanName

Syntax: `GlobalDefaultForSMOrigCallPlanName = "name"`
Description: This specifies the global default call plan for SM originating.
Type: string
Optionality:
Allowed:
Default: ""
Notes:
Example:

GlobalDefaultSMTermCallPlanName

Syntax: `GlobalDefaultForSMTermCallPlanName = "name"`
Description: This specifies the global default control plan for SM terminating.
Type: string
Optionality:

Allowed:
Default: ""
Notes:
Example:

globalCapabilityFlushPeriod

Syntax: `globalCapabilityFlushPeriod = seconds`
Description: Sets the flush period in seconds. This overrides the default (1 hour) CCS capability cache flush period.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default: 3600 (1 hour)
Notes: Enables updates to the default control plan to be recognized by the service loader more quickly.
Example: `globalCapabilityFlushPeriod = 10`

IncomingCallBarDisable

Syntax:
Description:
Type:
Optionality:
Allowed:
Default:
Notes: This parameter is not used.
Example:

IncomingCallBarEnable

Syntax:
Description:
Type:
Optionality:
Allowed:
Default:
Notes: This parameter is not used.
Example:

InterpretAccountNumberAsCLI

Syntax:
Description: Whether to interpret the subscriber number as a CLI.
Type:
Optionality:
Allowed: true, false
Default: false
Notes: This parameter is not used.
Example:

MobileTerminatingHomeCli

Syntax: `MobileTerminatingHomeCli = "cli"`

Description: Defines the CLI to use to replace the normalized calling number in the ACS Chassis when the service being used is 'Roaming'.

Type:

Optionality:

Allowed:

Default: ""

Notes:

Example: `MobileTerminatingHomeCli = ""`

NoCallPlanError

Syntax: `NoCallPlanError = "sev"`

Description: This is the severity of the syslog message when no control plan is found for the CCS service.

Type: Integer

Optionality:

Allowed: notice, warning, error, critical

Default: warning

Notes:

Example: `NoCallPlanError = "warning"`

PreCallAnnouncementId

Syntax: `PreCallAnnouncementId = id`

Description: This is the ID of the pre call announcement as used by the UATB node.

Type:

Optionality:

Allowed: A valid pre call announcement ID. This can be any entry ID from the announcements table.

Note: This ID cannot be viewed from any announcement configuration screen.

Default: 0

Notes: A zero setting indicates there is no pre call announcement.

Example: `PreCallAnnouncementId = 0`

PreCallLowBalance

Syntax: `PreCallLowBalance = true|false`

Description: Determines whether or not to enable pre-call low balance warnings.

Type: Boolean

Optionality:

Allowed: false, true

Default: false

Notes: This parameter is used by the UATB node in conjunction with the `WithheldDuration` parameter.

Example: `PreCallLowBalance = false`

productCapabilitiesCacheFlushTime

Syntax: `productCapabilitiesCacheFlushTime = seconds`

Description: How often a check is made for data older than its validity time.

Type: Integer

Optionality: Optional (default used if not set)

Allowed: Any positive integer

Default: 120

Notes: Applies to product capabilities matched on capability and product type. To reload the cache more frequently with the latest versions of control plans, set `productCapabilitiesCacheFlushTime` to a low value. For example, when set to 60, the cache is flushed every 60 seconds. The value should be less than or equal to that of `callPlanAndDataCacheFlushTime` so that a valid capability is used when retrieving control plan data.

Example: `productCapabilitiesCacheFlushTime = 60`

productCapabilitiesCacheMaxAge

Syntax: `productCapabilitiesCacheMaxAge = seconds`

Description: The time after which an unused or unchanged product capability is dropped from cache.

Type: Integer

Optionality: Optional (default used if not set)

Allowed: Any positive integer

Default: 3600

Notes: Applies to product capabilities matched on capability and product type. To remove stale entries from cache more frequently, set `productCapabilitiesCacheMaxAge` to a low value. For example, when set to 900, ccsSvcLibrary removes entries that have been unused/unchanged for 900 seconds.

The value should be greater than that of `productCapabilitiesCacheValidityTime` and less than or equal to that of `callPlanAndDataCacheMaxAge`.

Example: `productCapabilitiesCacheMaxAge = 900`

productCapabilitiesCacheValidityTime

Syntax: `productCapabilitiesCacheValidityTime = seconds`

Description: The maximum age of the data before it is refreshed from the database

Type: Integer

Optionality: Optional (default used if not set)

Allowed: Any positive integer

Default: 600

Notes: Applies to product capabilities matched on capability and product type. To refresh entries in the cache more frequently, set `productCapabilitiesCacheValidityTime` to a low value. For example, when set to 300, entries are refreshed after 300 seconds by the next flushing cycle.

The value should be greater than that of `productCapabilitiesCacheFlushTime` and less than or equal to that of `callPlanAndDataCacheValidityTime`.

Example: `productCapabilitiesCacheValidityTime = 300`

productTypeForExternalSub

Syntax: `productTypeForExternalSub = "pt_name"`

Description: Specifies the name of the product type for external subscribers that do not exist on the NCC platform. NCC uses the product type for external subscribers when sending requests to update external subscriber balances to Oracle Communications Billing and Revenue Management (BRM) Elastic Charging Engine (ECE) through a Diameter interface.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: "EXTERNAL"

Notes:

Example: `productTypeForExternalSub = "EXTERNAL"`

promptForAccountOnOriginatingSK

Syntax: `promptForAccountOnOriginatingSK = true|false`

Description: When set to true, the service library will prompt the caller to enter the subscriber number and PIN when:

- The ccsSvcLibrary cannot identify the subscriber who is calling
- The call was not triggered with an INAP service key associated with the service handle of "CCS_ROAM" or "SM_MT" in the **SLEE.cfg** file.

Type: Boolean

Optionality:

Allowed: true, false

Default: true

Notes:

Example: `promptForAccountOnOriginatingSK = true`

promptForAccountOnTerminatingSK

Syntax: `promptForAccountOnTerminatingSK = true|false`

Description: When set to true, the service library will prompt the caller to enter the subscriber number and PIN when the:

- ccsSvcLibrary cannot identify the subscriber who is calling
- Call was triggered with an INAP service key associated with the service handle of "CCS_ROAM" or "SM_MT" in the **SLEE.cfg** file.

Type: Boolean

Optionality:

Allowed: true, false

Default: true

Notes:

Example: `promptForAccountOnTerminatingSK = true`

RetrieveLCRNumbers

Syntax: `RetrieveLCRNumbers = true|false`

Description: Determines whether the UATB node can retrieve LCR numbers.

Type: Boolean

Optionality:
Allowed: true, false
Default: true
Notes:
Example: `RetrieveLCRNumbers = true`

SingleReservation

Syntax: `SingleReservation = true|false`
Description: Switches single reservation on or off.
Type: Boolean
Optionality:
Allowed: true, false
Default: false
Notes:
Example: `SingleReservation = false`

SubcriberDomainType

Syntax: `SubscriberDomainType = id`
Description: The ID of the domain type through which subscribers are stored (normally the VWS).
Type: Integer
Optionality: Optional (default used if not set).
Allowed: A valid domain type ID, as defined in a CCS domain type on the **Domain** tab in the Service Management screen.
Default: 1 (for VWS)
Notes: For more information about domains, see *Domains* (on page 10).
For more information about what ID corresponds to the domain type which is used for an application, see the application's technical guide.
Example: `SubscriberDomainType = 1`

UnknownDataReleaseCause

Syntax: `UnknownDataReleaseCause = int`
Description: Defines the release cause to send back to the switch in the TCAP-CONNECT when the service cannot be loaded.
Type: Integer
Optionality:
Allowed:
Default: 31
Notes:
Example: `UnknownDataReleaseCause = 31`

VoucherDomainType

Syntax: `VoucherDomainType = id`
Description: The ID of the domain type through which vouchers are redeemed (normally the VWS).
Type: Integer

Optionality:	Optional (default used if not set).
Allowed:	A valid domain type ID, as defined in a CCS domain type on the **Domain** tab in the Service Management screen.
Default:	1 (for VWS)
Notes:	2 sets voucher redemptions to process through the Voucher Manager server. When the CCS Balance Top Up Suite is installed, the `VoucherDomainType` is automatically set to "2". You can manually change the value back to '1' to use the VWS even when the CCS Balance Top Up Suite SLC package is installed. For more information about domains, see Overriding default domain types. For more information about what ID corresponds to the domain type which is used for an application, see the application's technical guide.
Example:	`VoucherDomainType = 1`

`WithheldDuration`

Syntax:	`WithheldDuration = seconds`
Description:	The length of time withheld for low balance warnings.
Type:	Integer
Optionality:	
Allowed:	
Default:	0
Notes:	This parameter is used by the UATB node.
Example:	`WithheldDuration = 0`

libccsClientPlugins

Purpose

libccsClientPlugins is a library which provides CCS plug-ins to the beClient. The plug-ins include:

- VoucherRechargePlugin
- VoucherTypeRechargePlugin
- MergeWalletsPlugin

Startup

libccsClientPlugins is used if the library and one or more of its functions is included in a `plugins` section in **eserv.config**. For an example of a process which uses this library, see *plugins* (on page 176).

Configuration

libccsClientPlugins is configured in the section specified in the config parameter in the plug-ins entry which calls the related function and the libccsClientPlugins library.

For examples, see *plugins* (on page 176).

`voucherRechargeOptions`

Name of the configuration section required for the Voucher Recharge plug-in.

sendBadPin

Syntax: `sendBadPin = true|false`

Description: When true, increments the Bad PIN for a failed voucher recharge.

Type:

Optionality:

Allowed: true, false

Default: false

Notes: Used for invalid voucher number or voucher PIN only - does not apply to failed wallet recharges.

Example: `sendBadPin = false`

singleBonusEdrs

Syntax: `singleBonusEdrs = true|false`

Description: Whether to produce a single bonus EDR.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

Default: false

Notes:

Example: `singleBonusEdrs = false`

srActivatesPreuseAccount

Syntax: `srActivatesPreuseAccount = true|false`

Description: Weather or not SR (Voucher Recharge) activate wallets with a Pre-use state.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Voucher recharges can activate pre-use wallets.
false	Voucher recharges cannot be used with pre-use wallets.

Default: false

Notes: The application of this parameter is also affected by *rechargePreUseAccounts* (on page 294).

Example: `srActivatesPreuseAccount = false`

srasActivatesPreuseAccount

Syntax: `srasActivatesPreuseAccount = true|false`

Description: When true, SRAS activates the wallet.

Type: Boolean

Optionality:

Allowed: true, false

Default: false

Notes:

Example: `srasActivatesPreuseAccount = false`

`voucherRechargeTriggers`

This configuration is required for the Voucher plug-in. It defines the type of message that triggers the plug-in.

libccsCommon

Purpose

libccsCommon provides common functions to various CCS processes.

Startup

libccsCommon is used by a number of CCS processes. No startup configuration is required for this library to be used.

Configuration

The libccsCommon library supports parameters from the common parameter group in the **eserv.config** file on all machines. For more information, see Configuration.

Chapter 5

Background Processes on the VWS

Overview

Introduction

This chapter provides a description of the programs or executables used by CCS as background processes on the VWSs.

Executables are located in the **/IN/service_packages/CCS/bin** directory.

Some executables have accompanying scripts that run the executables after performing certain cleanup functions. All scripts should be located in the same directory as the executable.

For more information about the processes and systems that use these programs and executables, see *System Overview* (on page 1).

Warning: It is a prerequisite for managing these core service functions that the operator is familiar with the basics of Unix process scheduling and management. Specifically, the following Unix commands:

- init (and inittab)
- cron (and crontab)
- ps
- kill

In this chapter

This chapter contains the following topics.

beVWARS 210
ccsActivationCharge 212
ccsBadPinPlugin 213
ccsCB10HRNAES 214
ccsCB10HRNSHA 215
ccsBeAvd 215
ccsExpiryMessageGenerator 215
ccsLegacyPIN 216
ccsMFileCompiler 216
ccsNotification 220
ccsSLEEChangeDaemon 223
ccsPDSMSPlugin 232
ccsRewardsPlugin 234
ccsPMXPlugin 238
ccsVWARSActivation 241
ccsVWARSAmountHandler 243
ccsVWARSExpiry 244
ccsVWARSNamedEventHandler 252
ccsVWARSPeriodicCharge 257
ccsVWARSQuota 262
ccsVWARSRechargeHandler 263
ccsVWARSReservationHandler 263
ccsVWARSVoucherHandler 270
ccsVWARSWalletHandler 274
ccsVWARSPolicyHandler 276
ccsWLCPlugin 276
ccsPolicyPlugin 277
cmnPushFiles 277
libccsCommon 282
libccsVWARSUtils 292

beVWARS

Purpose

beVWARS is the Vouchers Wallets Accounts Reserve System. It enables CCS to handle actions that interact with the wallet, account, and voucher tables in the E2BE database on the VWS. Most beVWARS functionality is provided by plug-ins and handlers as defined in the *handlers* (on page 211) and *plugins* (on page 212) parameters. This section shows beVWARS configuration, which includes CCS plug-ins and handlers.

Note: If the VWS is not used, the beVWARS handlers and plug-ins are not relevant.

Example

An example of the beVWARS parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
beVWARS = {
    other beVWARS configuration

    handlers = [
```

```
        VWS beVWARS handlers

        "ccsVWARSReservationHandler.so"
        "ccsVWARSNamedEventHandler.so"
        "ccsVWARSRechargeHandler.so"
        "ccsVWARSAmountHandler.so"
        "ccsVWARSWalletHandler.so"
        "ccsVWARSPolicyHandler.so"
    ]
    plugins = [
        VWS beVWARS plug-ins

        "ccsVWARSExpiry.so"
        "ccsRewardsPlugin.so"
        "ccsVWARSActivation.so"
        "ccsPDSMSPlugin.so"
        "ccsNotification.so"
        "ccsWLCPlugin.so"
        "ccsBadPinPlugin.so"
        "ccsPMXPlugin.so"
        "ccsPolicyPlugin.so"
    ]
}
```

Note: Other handlers and plug-ins may be provided which extension features (for example the ccsVWARSVoucherHandler is provided by the Voucher Manager feature). For more information about those libraries, see the documentation provided with the feature.

Parameters

beVWARS has two parameters which are relevant to CCS configuration. They are documented below. For more information about other beVWARS parameters, see *VWS Technical Guide*.

`handlers`

Syntax:

```
handlers = [
    "lib"
    [...]
 ]
```

Description: Lists the beVWARS message handler plug-ins to load.

Type: Array

Optionality: Required to load handlers which handle messages from CCS processes such as *ccsBeOrb* (on page 89).

Allowed:

Default:

Notes: This array will also include the standard handlers provided by VWS, and may also include plug-ins from other applications such as Promotion Manager.

For more information about the standard handlers provided with CCS including their configuration, see the following:

- *ccsVWARSReservationHandler* (on page 263)
- *ccsVWARSNamedEventHandler* (on page 252)
- *ccsVWARSRechargeHandler* (on page 263)
- *ccsVWARSAmountHandler* (on page 243)
- *ccsVWARSWalletHandler* (on page 274)
- *ccsVWARSPolicyHandler* (on page 276)

Example:

```
handlers = [
    "ccsVWARSReservationHandler.so"
```

```
    "ccsVWARSNamedEventHandler.so"
    "ccsVWARSRechargeHandler.so"
    "ccsVWARSAmountHandler.so"
    "ccsVWARSWalletHandler.so"
    "ccsVWARSPolicyHandler.so"
]
```

`plugins`

Syntax:

```
plugins = [
    "lib"
    [...]
]
```

Description: Lists the beVWARS event plug-ins to load.

Type: Array

Optionality: Required to load event plug-ins which perform functions needed by CCS.

Allowed:

Default:

Notes: Where plug-ins are triggered by the same event, they will operate in the order they appear in this list.

This array will also include the standard plug-ins provided by VWS, and may also include plug-ins from other applications such as Promotion Manager.

For more information about the standard plug-ins provided with CCS including their configuration, see the following:

- *ccsVWARSExpiry* (on page 244)
- *ccsRewardsPlugin* (on page 234)
- *ccsVWARSActivation* (on page 241)
- *ccsPDSMSPlugin* (on page 232)
- *ccsNotification* (on page 220)
- *ccsWLCPlugin* (on page 276)
- *ccsBadPinPlugin* (on page 213)
- *ccsPMXPlugin* (on page 238)
- *ccsPolicyPlugin* (on page 277)

Example:

```
plugins = [
    "ccsVWARSExpiry.so"
    "ccsRewardsPlugin.so"
    "ccsVWARSActivation.so"
    "ccsPDSMSPlugin.so"
    "ccsNotification.so"
    "ccsWLCPlugin.so"
    "ccsBadPinPlugin.so"
    "ccsPMXPlugin.so"
    "ccsPolicyPlugin.so"
]
```

ccsActivationCharge

Purpose

ccsActivationCharge is a beVWARS plug-in which:

- Processes wallets as they activate (triggers on a wallet activated event)
- Applies any periodic charges which apply to the wallet and have Charge on Activation set to true.

For more information about periodic charge configuration, see *Charging Control Services User's Guide*.

Note: This process only applies to periodic charges which were configured in CCS 3.1.4 or earlier.

Startup

If ccsActivationCharge is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsActivationCharge.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Parameters

The ccsActivationCharge supports the following parameter in the `ccsActivationCharge` section of **eserv.config**.

`periodicChargeCacheValidityPeriod`

Syntax:	`periodicChargeCacheValidityPeriod = seconds`
Description:	Time out value in seconds for the periodic charge cache.
Type:	Integer
Optionality:	Optional
Allowed:	Any positive decimal integer.
Default:	600
Notes:	
Example:	`periodicChargeCacheValidityPeriod = 600`

Example

An example of the `ccsActivationCharge` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsActivationCharge = {
    periodicChargeCacheValidityPeriod = 600
}
```

ccsBadPinPlugin

Purpose

ccsBadPinPlugin is a beVWARS event plug-in that checks for bad PIN thresholds. It is triggered by a balance value changed event.

Startup

If ccsBadPinPlugin is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsBadPinPlugin.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Configuration

ccsBadPinPlugin supports the parameters from the `badPinPlugin` section of **eserv.config**.

Note: Some of the ccsVWARSVoucherHandler parameters are also used by ccsBadPinPlugin:

- *clearConsecutivePin* (on page 271)
- *dailyBadPinExpiryHours* (on page 272)
- *weeklyBadPinExpiryHours* (on page 272)
- *monthlyBadPinExpiryHours* (on page 272)
- *consecutiveBadPinExpiryHours* (on page 271)
- *vomsInstalled* (on page 273)

`cacheFlushPeriod`

Syntax:	`cacheFlushPeriod = seconds`
Description:	The number of seconds before refreshing the balance type cache used by ccsBadPinPlugin.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	200
Notes:	
Example:	`cacheFlushPeriod = 300`

`cacheValidityTime`

Syntax:	`cacheValidityTime = seconds`
Description:	The number of seconds an entry is kept before the entry's record is re-read.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	10
Notes:	
Example:	`cacheValidityTime = 30`

ccsCB10HRNAES

License

The ccsCB10HRNAES library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsCB10HRNSHA

License

The ccsCB10HRNSHA library is available only if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsBeAvd

License

The ccsBeAvd binary is only available if you have purchased the Voucher Management license.

For more information about this library, see *Voucher Manager Technical Guide*.

ccsExpiryMessageGenerator

Purpose

ccsExpiryMessageGenerator generates a list of wallets or balances which will expire shortly. The list of subscribers is generated on the VWSs and transfered to the SMS, where they are actioned by ccsExpiryMessageLoader.

Startup

The CCS install process adds the ccsExpiryMessageGenerator process to the crontab, running at 9 am on each day of month for ccs_oper by default.

It is scheduled as **/IN/service_packages/CCS/bin/ccsExpiryMessageGenerator** by the following line:

```
0 2 * * *    . /IN/service_packages/CCS/.profile ; .
/IN/service_packages/CCS/.profile-be ;
/IN/service_packages/CCS/bin/ccsExpiryMessageGenerator >>
/IN/service_packages/CCS/tmp/ccsExpiryMessageGenerator.log 2>&1
```

Parameters

Available parameters are detailed in *ccsExpiryMessageLoader* (on page 137).

Example

```
CCS = {
    ExpiryMessages = {
        walletExpiryPeriod = 15
        balanceExpiryPeriod = 10
        balanceTypes = [ 1 ]
        oracleUsername = ""
        oraclePassword = ""
        generatorFilename = "ccsExpiryMessages"

        generatorFiledir = "/IN/service_packages/CCS/logs/expiryMessageWrite/"
        inputDirectory = "/IN/service_packages/CCS/logs/expiryMessageRead/"

        cmnPushFiles = [
            "-d", "/IN/service_packages/CCS/logs/expiryMessageWrite/"
            "-r", "/IN/service_packages/CCS/logs/expiryMessageRead/"
```

```
            "-h", "produsms01"
            "-p", "2027"
            "-F"
        ]
    }
}
```

This section of the **eserv.config** must be set up on the SMS and VWS for expiry notification short messages sent from the ccsExpiryMessageGenerator and ccsExpiryMessageLoader processes. If this section is not present, then no expiry notifications will be sent at all.

Note: ccsExpiryMessageGenerator uses ORACLE_SID to connect to database. It is recommended to set ORACLE_SID to a valid TNS entry. In case ORACLE_SID is not set, then ccsExpiryMessageGenerator process establishes connection to database using the oracleUsername and oraclePassword provided in **eserv.config**.

Failure

If ccsExpiryMessageGenerator fails, no expiry notifications will be sent at all.

Output

The notification request files produced by ccsExpiryMessageGenerator are in the format:

```
notif_id lang_id MSISDN num_params param1[|param2|...]
```

ccsExpiryMessageGenerator writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsExpiryMessageGenerator.log**.

ccsLegacyPIN

Purpose

ccsLegacyPIN plug-in library is used by *ccsAccount* (on page 297) and the ccsVoucher_CCS3 voucher tool to encrypt the PINs using the DES authentication rule. For more information about authentication rules, see *Security libraries* (on page 23).

Note: The ccs3Encryption plug-in is a symbolic link to the *ccsLegacyPIN* (on page 142) plug-in, but in the ccs3Encryption mode it uses different parameters.

Startup

ccsLegacyPIN is used by ccsVoucher_CCS3 as necessary. No startup configuration is required for this library to be used.

Configuration

ccsLegacyPIN has no specific configuration. It does accept some parameters from ccsVoucher_CCS3 for voucher encryption which are configured in the CCS Voucher Management and Service Management screens.

ccsMFileCompiler

Purpose

MFiles store data that is not updated very often (for example, tariffing data). ccsMFIleCompiler compiles MFiles on the Voucher and Wallet Server to provide a fast lookup for the stored data.

When a new row is replicated into the CCS_MFILE table on the E2BE database, ccsMFileCompiler processes the tariffing or named event catalogue data in the E2BE database and creates an MFile for the VWS processes to use.

For more information about MFile processing, see the discussion on MFile updates in *VWS Technical Guide.* For information about MFile configuration, see the section on MFile generation in *Charging Control Services User's Guide*.

MFile filenames

ccsMFileCompiler generates MFile filenames based on the service provider ID and the date and time that the MFile is created. For rating MFiles, ccsMFileComplier use the following format:

*acs_Cust_ID*D*timestamp*

For named event catalogue MFiles, ccsMFileCompiler uses the following format:

P*acs_Cust_ID*D*timestamp*

where *acs_Cust_ID* is the ID of the service provider in the ACS_CUST_ID field of the CCS_MFILE table, and *timestamp* is the date and time when ccsMFileCompiler created the file. For example, the following rating MFile would be for a service provider with ID **11**:

11D20150330110120

Note: For backward compatibility, if *acs_Cust_ID* is **0** (zero), then ccsMfileCompiler generates the filename using only the *timestamp.* For example, the filename format is "*timestamp*" for rating MFIles or "P*timestamp*. for named event catalogue MFiles. For example, the following rating MFIle would be for a service provider with ID **0**:

20150330110120

Startup

ccsMFileCompiler is started by entry ccs9 in the inittab, through the **/IN/service_packages/CCS/bin/ccsMFileCompilerStartup.sh** shell script.

Configuration

ccsMFileCompiler reads the following configuration from the CCS and BE sections of the **eserv.config** file:

```
CCS = {
    oracleUserAndPassword = "user/pwd"

    MFile = {
        path = "dir"
        numberOfErrors = int
        timeout = int
    }

BE = {
    serverId = int
    amPrimary = true|false
    beLocationPlugin = "lib"
    }
```

Parameters

This section describes the ccsMFileCompiler configuration parameters in the CCS section of the **eserv.config** file.

ccsMFileCompiler uses the `oracleUserAndPassword` parameter from the CCS section of **eserv.config** to retrieve Oracle database login details. For more information, see *oracleUserAndPassword* (on page 52).

MFile Configuration Parameters

ccsMFileCompiler supports the following parameters from the CCS, MFile section of **eserv.config**:

`path`

Syntax:	`path = "dir"`
Description:	The location of compiled MFiles.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	"/IN/service_packages/CCS/MFile"
Notes:	
Example:	`path = "/var/CCS/MFile"`

`numberOfErrors`

Syntax:	`numberOfErrors = int`
Description:	The number of compile errors that can occur before the ccsMFileCompiler process will stop.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1
Notes:	
Example:	`numberOfErrors = 1`

`timeout`

Syntax:	`timeout = microsecs`
Description:	The number of microseconds to wait to successfully connect to the beServer before timing out.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	20000
Notes:	
Example:	`timeout = 5000`

Example MFile Configuration

The following shows an example MFile configuration section of a **eserv.config** file on the Voucher and Wallet Server.

```
MFile = {
    path = "/IN/service_packages/CCS/MFile"
    numberOfErrors = 1
    timeout = 20000
}
```

Shared Configuration Parameters

ccsMFileCompiler uses the following shared parameters defined in the `BE` section of **eserv.config** to retrieve details of the Voucher and Wallet Server to which it should connect, and to reload the MFile data:

- `amPrimary`
- `serverId`
- `beLocationPlugin`

For information about configuring BE shared parameters, see *BE* **eserv.config** *parameters* (on page 135).

ccsMFileCompiler Command Line Parameters

ccsMFileCompiler supports the following optional command line parameters:

```
ccsMFileCompiler [-r row_id] [-l be_location_plugin] [-a true|false] [-i
be_server_id] [-d debug_flag]
```

Parameters

`-r`

Syntax:	`-r row_id`
D3escription:	Runs the ccsMFileCompiler process for a specific row in the CCS_MFILE table, where *row_id* identifies the row for which the the process should be run.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	None
Notes:	Runs ccsMFileCompiler once and then exits.
Example:	`-r 10`

`-l`

Syntax:	`-l be_location_plugin`
Description:	Specifies the location of the BE plug-in. This value overrides the value configured for the `beLocationPlugin` parameter in the `BE` section of **eserv.config**.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	libGetccsBeLocation.so
Notes:	
Example:	`-l "libGetccsBeLocation.so"`

`-a`

Syntax:	`-a true|false`
Description:	Set to true if this is the primary VWS. Otherwise set to false.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true false

Default:	true
Notes:	Overrides the value configured for amPrimary in the `BE` section of **eserv.config**.
Example:	`-a true`

`-i`

Syntax:	`-i be_server_id`
Description:	Sets the ID of the VWS pair where *`be_server_id`* is the ID of the VWS.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	1
Notes:	Overrides the value configured in the `serverId` parameter in the `BE` section of **eserv.config**.
Example:	`-i 1`

`-d`

Syntax:	`-d debug_flag`
Description:	Defines which flag you want to use for debugging.
Type:	String
Optionality:	Optional
Allowed:	all – full debugging ccsMFileCompiler – component only debugging none – no debug
Default:	None
Notes:	
Example:	`-d all`

Failure

If ccsMFileCompiler fails, MFile updates will stop.

MFile entries will still be replicated to the CCS_MFILE table in the E2BE database, but they will not be processed. The corresponding MFile will not be created for the unprocessed entries and therefore beVWARS will not use any rating or named event catalogue changes made since the last MFile was created.

Output

ccsMfileCompiler writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsMFileCompiler.log**.

ccsNotification

Purpose

ccsNotification is a beVWARS event plug-in that generates a list of real-time wallet notifications for delivery. Notifications can be triggered on the following events:

- Wallet expiry

- Wallet state change
- Balance value changed
- Bucket expiry

Note: Other plug-ins, such as ccsVWARSPeriodicCharge, can write notifications. For more information about notifications and events that trigger notifications, see *Notifications* (on page 39).

Real-time Wallet Notifications Delivery Process

The following high-level process describes how the ccsNotification process delivers real-time wallet notifications. For more information, see *Real-Time Notifications* (on page 367).

1. When a wallet or bucket is triggered through beVWARS on a primary VWS, ccsNotification checks whether a real-time wallet notification should be sent.

   The criteria for sending real-time wallet notifications and the templates they are based on are defined in the **Prepaid Charging**, **Wallet Management** window in the CCS user interface (UI), and replicated to the VWS. For more information about configuring real-time wallet notifications, see *Charging Control Services User's Guide*.
2. ccsNotification checks the E2BE database to establish whether the real-time notification uses an ACS template or a DAP template. For information about how real-time wallet notifications which are based on DAP templates are delivered, see *DAP Notification Delivery* (on page 368).
3. ccsNotification looks up the text configured for the template in the database and creates the final notification text by substituting values for any variables. For information about configuring ACS notification templates, see *Advanced Control Services User's Guide*.
4. ccsNotification delivers the notification through the beServiceTrigger process.

Processes Used to Deliver Real-time Wallet Notifications

This table lists the main processes involved in sending real-time wallet notifications for delivery.

Process	Role	Further information
ccsNotification	ccsNotification is a beVWARS event plug-in that generates a list of real-time wallet notifications for delivery.	
beVWARS	beVWARS is the main VWS process that supports the ccsNotification plug-in and handles interaction with the E2BE database.	*VWS Technical Guide*
beServiceTrigger	Delivers the notification to the subscriber.	*VWS Technical Guide*

Startup

If ccsNotification is included in the beVWARS `plugins` array in **eserv.config**, beVWARS loads it during initialization.

To include ccsNotification in the beVWARS plugins configuration, use the following syntax:

```
plugins = [
    "ccsNotification.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Configuration

The ccsNotification beVWARS plug-in is configured by the `notificationPlugin` parameter group in the **eserv.config** file on the VWS:

```
notificationPlugin = {
    xmlInterfaceName = "name"
    cacheFlushPeriod = seconds
    cacheValidityTime = seconds
    useOldestExpiry = true|false
    UTCOffsetHours = hours
}
```

Parameters

ccsNotification plugin supports these parameters in the notificationPlugin section of **eserv.config**.

xmlInterfaceName

Syntax:	`xmlInterfaceName = "name"`
Description:	The name of the SLEE xml interface.
Type:	String
Optionality:	Required
Allowed:	
Default:	xmlIF
Notes:	
Example:	`xmlInterfaceName = "xmlIF"`

cacheFlushPeriod

Syntax:	`cacheFlushPeriod = seconds`
Description:	Sets the number of seconds between each clearance of the notification caches.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	
Default:	200
Notes:	
Example:	`cacheFlushPeriod = 200`

cacheValidityTime

Syntax:	`cacheValidityTime = seconds`
Description:	The length of time, in seconds, an entry is kept before the entry's record is re-read.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	10
Notes:	
Example:	`cacheValidityTime = 10`

`useOldestuseOldestExpiry`

Syntax:	`useOldestExpiry = true|false`
Description:	When a subscriber's balance contains multiple buckets, this parameter specifies which bucket's expiration date to include in the real-time wallet notification.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true - Uses the bucket with the expiration date set the furthest in the future. For example, if Bucket A expires 1 Jan 2016, and Bucket B expires 1 Sep 2016, the real-time wallet notification includes Bucket B's expiration date. false - Uses the bucket that expires first. For example, if Bucket A expires 1 Jan 2016, and Bucket B expires 1 Sep 2016, the real-time wallet notification includes Bucket A's expiration date.
Default:	true
Notes:	
Example:	`useOldestExpiry = true`

`UTCOffsetHours`

Syntax:	`UTCOffsetHours = hours`
Description:	For use in non-GMT/UTC time zones. The number of hours offset from Universal Coordinated Time (UTC) that are applied to wallet and balance expiry notifications. The CCSNotification plug-in converts this parameter to seconds and applies the offset to all timestamp variables in wallet and balance expiry notifications.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	+ or - the number of hours. For example, +5 or -5.
Default:	0
Notes:	
Example:	`UTCOffsetHours = +4`

Example

This text shows an example ccsNotification configuration.

```
notificationPlugin = {
    xmlInterfaceName = "xmlIF"
    cacheFlushPeriod = 200
    cacheValidityTime = 10
    useOldestExpiry = true
    UTCOffsetHours = +3
}
```

ccsSLEEChangeDaemon

Purpose

The ccsSLEEChangeDaemon has three main functional areas:

- Update assignment of periodic charges to wallets. The ccsSLEEChangeDaemon handles periodic charge changes such as a periodic charge being:
  - Added to CCS or being assigned to a product type

  - Removed from a product type or from CCS
- Update assignment of Wallet Life Cycle Plans to wallets. The ccsSLEEChangeDaemon handles WLC changes such as a WLC plan being:
  - Added to CCS or being assigned to a product type
  - Removed from a product type or from CCS
- It also handles balance expiry extensions, updating the balance types in the affected wallets by the defined extension configuration.

ccsSLEEChangeDaemon is a SLEE process which runs on the primary VWS node.

The daemon receives its tasks by reading CCS_PC_QUEUE table, which is hosted in the SMF database on the SMS and is replicated to the E2BE database on the VWS.

Startup

On start-up, ccsSLEEChangeDaemon finds the `-r` flag and will check for a node ID and run in primary VWS mode.

In order to start, ccsSLEEChangeDaemon must be referenced in the **SLEE.cfg** file. See Editing the SLEE.cfg.

Note: If the daemon is started on a secondary VWS VWS it will immediately shut down.

Configuration

ccsSLEEChangeDaemon supports parameters from the `ccsSLEEChangeDaemon` parameter group in the **eserv.config** file on a Voucher and Wallet Server. It contains parameters arranged in the structure shown in the example below.

```
ccsSLEEChangeDaemon = {

    # BE Client section. Mandatory.
    beClient = {
        # pollPeriod = 300

        # throttle = 1000

        # numCursorRows = 1000

        clientName = "be1_ccsSLEEChangeDaemon"

        # heartbeatPeriod = 30000000

        # connectionRetryTime = 5

        # messageTimeoutSeconds = 2

        # billingEngines = [
        #  { id = 1, # pair ID
        #    primary   = { ip="PRIMARY_BE_IP", port=1500 },
        #    secondary = { ip="SECONDARY_BE_IP", port=1500 }
        #   }
        # ]

        # serviceTriggerTimeout = 5

    } # beClient
} # ccsSLEEChangeDaemon
```

eserv.config parameters

ccsSLEEChangeDaemon supports the following parameters from the `CCS` section of **eserv.config**.

`beClient`

Syntax: `beClient = { config }`

Description: The configuration for the connection to the beServer on the VWS.

Type: Parameter group

Optionality: Mandatory

Allowed:

Default:

Notes: This configuration is for the libBeClientIF library which ccsSLEEChangeDaemon uses to manage the connection.

For more information about this library, see *VWS Technical Guide*.

Example:

`billingEngines`

Syntax:

```
billingEngines = [
     {   id = id
         primary   = { ip="ip", port=port },
         secondary = { ip="ip", port=port }
     }
 ]
```

Description: Overrides connection details that beLocationPlugin obtains from the database.

For more information about the parameters included in the array, see *billingEngines* (on page 92) configuration for the ccsBeOrb process.

Type: Array.

Optionality: Optional.

Allowed:

Default:

Notes: Identifies the Voucher and Wallet Servers and assigns their Internet connection details.

Include this section to ensure that ccsSLEEChangeDaemon only connects to the local domain. If omitted, ccsSLEEChangeDaemon will connect to all VWS domains.

Example:

```
billingEngines = [
     {   id = CHANGE_ME,
         primary   = { ip="PRIMARY_BE_IP", port=1500 },
         secondary = { ip="SECONDARY_BE_IP", port=1500 }
     }

]
```

`clientName`

Syntax: `clientName = "name"`

Description: The unique client name of the process.

Type: String

Optionality: Mandatory

Allowed: Must be unique.

Default: ccsSLEEChangeDaemon

Notes: If more than one client connects with the same name the BE server will drop the other, therefore name must be unique.

Example: `clientName = "be1_ccsSLEEChangeDaemon"`

connectionRetryTime

Syntax: `connectionRetryTime = seconds`

Description: The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.

Type: Integer

Optionality: Required

Allowed:

Default: 5

Notes: This parameter is used by libBeClientIF.

Example: `connectionRetryTime = 2`

heartbeatPeriod

Syntax: `heartbeatPeriod = microsecs`

Description: The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.

Type: Integer

Optionality: Optional (Default used if not present)

Allowed:

0	Disable heartbeat detection.
positive integer	Heartbeat period.

Default: 30000000

Notes: 1 000 000 microseconds = 1 second.

Example: `heartbeatPeriod = 30000000`

throttle

Syntax: `throttle = num`

Description: The maximum number of Voucher and Wallet Server updates per second.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

0	Disable throttling (no limit).
positive integer	Update limit.

Default: 1000

Notes:

Example: `throttle = 1000`

maxOutstandingMessages

Syntax: `maxOutstandingMessages = num`

Description: The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.

Type: Integer

Optionality: Required

Allowed:

Default: If this parameter is not set, the maximum is unlimited.

Notes: If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls.

The messages are queued until the Voucher and Wallet Server has reduced its outstanding load.

This parameter is used by libBeClientIF.

Example:
```
maxOutstandingMessages = 100
```

messageTimeoutSeconds

Syntax:
```
messageTimeoutSeconds = seconds
```

Description: The time that the client process will wait for the server to respond to a request.

Type: Integer

Units: Seconds

Optionality: Required

Allowed:

1-604800	Number of seconds to wait.
0	Do not time out.

Default: 2

Notes: After the specified number of seconds, the client process will generate an exception and discard the message associated with the request.

This parameter is used by libBeClientIF.

Example:
```
messageTimeoutSeconds = 2
```

numCursorRows

Syntax:
```
numCursorRows = num
```

Description: The maximum number of cursor rows processed on the VWS when doing balance extensions.

Type: Integer

Optionality: Optional (default used if not set).

Allowed: Any number between 100 and 1000000. The closest number divisible by 100 will be used.

Default: 1000

Notes:

Example:
```
numCursorRows = 1000
```

plugins

Syntax:
```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    ...
]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Optional (as plug-ins will not be loaded if they are not configured here, this parameter must include any plug-ins which are needed to supply application functions; for more information about which plug-ins to load, see the BeClient section for the application which provides the BeClient plug-ins).

Allowed:

Default: Empty (that is, do not load any plug-ins).

Notes: The libclientBcast plug-in must be placed last in the plug-ins configuration list.

For more information about the libclientBcast plug-in, see *VWS Technical Guide*.

This parameter is used by libBeClientIF.

Example:

```
plugins = [
    {
        config="broadcastOptions"
        library="libclientBcast.so"
        function="makeBroadcastPlugin"
    }
]
```

config

Syntax: `config="name"`

Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `config="voucherRechargeOptions"`

function

Syntax: `function="str"`

Description: The function the plug-in should perform.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `function="makeVoucherRechargePlugin"`

library

Syntax: `library="lib"`

Description: The filename of the plug-in library.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `library="libccsClientPlugins.so"`

`reportPeriodSeconds`

Syntax:	`reportPeriodSeconds = seconds`
Description:	The number of seconds separating reports of failed messages.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	
Default:	10
Notes:	BeClient issues a failed message report: • For timed-out messages • For unrequested responses • For new calls rejected because of congestion • For messages with invalid Voucher and Wallet Server identifiers • If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`. This parameter is used by libBeClientIF.
Example:	`reportPeriodSeconds = 10`

`serviceTriggerTimeout`

Syntax:	`serviceTriggerTimeout = seconds`
Description:	The maximum duration, in seconds, the change daemon waits for beServiceTrigger response when control plans are triggered through the OSD interface
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	5
Notes:	
Example:	`serviceTriggerTimeout = 5`

BE eserv.config parameters

The following parameters are available in the `BE` section of the **eserv.config**.

`amPrimary`

Syntax:	`amPrimary = true|false`
Description:	True if this is the primary VWS in the pair.
Type:	Boolean
Optionality:	Optional, default used if not set
Allowed:	
Default:	true
Notes:	
Example:	`amPrimary = false`

`beLocationPlugin`

Syntax:	`beLocationPlugin = "lib"`
Description:	The plug-in library that finds the Voucher and Wallet Server details of the Voucher and Wallet Servers to connect to.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	libGetccsBeLocation.so
Notes:	This library must be in the LD_LIBRARY_PATH.
Example:	`beLocationPlugin = "libGetccsBeLocation.so"`

`serverId`

Syntax:	`serverId = id`
Description:	The ID of the VWS pair.
Type:	Integer
Optionality:	
Allowed:	
Default:	1
Notes:	Set to 1 if this is not a VWS
Example:	`serverId = 11`

Command line parameter

ccsSLEEChangeDaemon supports the following command-line switch.

```
ccsSLEEChangeDaemon -r id
```

`-r`

Syntax:	`-r id`
Description:	The node ID of the VWS node on which the ccsSLEEChangeDaemon is running.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	If not set, will not start.
Notes:	Node number must be between 512 and 1023.
Example:	`-r 531`

Failure

While ccsSLEEChangeDaemon is down, periodic charge assignment updates will not be executed on the local machine. In addition, wallet updates for balance expiry extensions will not be processed.

This table describes the recovery and failure files used by ccsSLEEChangeDaemon to attempt to recover after a failure.

File	Details
.recovery	These files are only written on the VWS VWS. They have the following naming convention: `.recovery.ACSCustomerID.CCS_PC_QUEUE.ID` These files are written for a every 100th row processed and also on VWS "No Connection" error. The file should contain one line. For periodic charge updates it will contain: *SubscriberId*|*WalletId* For balance expiry extensions it will contain: *WalletId* Each time ccsSLEEChangeDaemon writes one of these files, it will also raise a Warning level alarm. If ccsSLEEChangeDaemon fails while processing a batch, it will reprocess CCS_PC_QUEUE from the point recorded in the .recovery file. These files are automatically deleted by ccsSLEEChangeDaemon.
.failed	These files are written on both the SMS and the VWS. They have the following naming convention: `.failed.<ACS Customer ID>.<CCS_PC_QUEUE.ID>` An entry is written to this file for each wallet update which initially fails. They contain a line for each failure: • For periodic charge and WLC updates: *SubscriberId*|*WalletId*|*PeriodicChargeBalanceTypeId*|*ProductId*|*ChangeAction* • For balance expiry extensions: *SubscriberId*|*WalletId*|*PeriodicChargeBalanceTypeId*|*PCQProductId*|*PCQNumMonths*|*PCQNumDays* Each time ccsSLEEChangeDaemon adds an entry to this file, it will also raise an Error level alarm. ccsSLEEChangeDaemon reads the entries in this file and attempts to reprocess them. Once all the entries in the file have been reprocessed, the ccsSLEEChangeDaemon deletes them.

failed	These files are written on both the SMS and the VWS. They have the following naming convention:
	`failed.<ACS Customer ID>.<CCS_PC_QUEUE.ID>`
	An entry is written to this file every time an entry in the .failed file is resent, and fails a second time. This file's first two lines are:
	`# Periodic Charge Change Daemon: failed updates`
	`#`
	SubscriberId|*WalletId*|*PeriodicChargeBalanceTypeId*|*ChangeType*|*ChangeAction*|*DomainId*|*NumberofBalanceTypes*[|*BalanceTypeId*|*BucketId*|*BucketValue*[|...]]
	Then there is an entry for each wallet update which fails a second time:
	SubscriberId|*WalletId*|*PeriodicChargeBalanceTypeId*|*ChangeType*|*ChangeAction*|*DomainId*|*NumberofBalanceTypes*[|*BalanceTypeId*|*BucketId*|*BucketValue*[|...]]
	For balance expiry extensions on the VWS VWS the entry is:
	SubscriberId|*WalletId*|*PeriodicChargeBalanceTypeId*|*PCQProductId*|*PCQNumMonths*|*PCQNumDays*
	Each time ccsSLEEChangeDaemon writes an entry to this file, it will raise an Error level alarm.
	failure files are left for manual recovery.

Note: If an operation fails due to a "No Connection" error, ccsSLEEChangeDaemon will raise a LOGGED_WARNING and stop processing the row.

Output

ccsSLEEChangeDaemon writes recovery and failure logs for period charge updates to **/IN/service_packages/CCS/logs/ccsSLEEChangeDaemon/ccsPCChange/**.

ccsSLEEChangeDaemon writes recovery and failure logs for balance expiry extensions to **/IN/service_packages/CCS/logs/ccsSLEEChangeDaemon/ccsBalExtension/**.

If one of these files cannot be written to, the ccsSLEEChangeDaemon will exit with a critical error (for alarm details, see *CCS Alarms Reference Guide*).

ccsSLEEChangeDaemon writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsSLEEChangeDaemon.log**.

ccsPDSMSPlugin

Purpose

ccsPDSMSPlugin handles the promotional destination of notifications. The configuration identifies the balance type that holds the number of promotional notifications sent by the customer.

It is triggered by wallet activated and bucket expiry events.

Startup

If ccsPDSMSPlugin is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsPDSMSPlugin.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Parameters

Parameters for ccsPDSMSPlugin are contained in the `ccsPromotionalDestinationSMS` section of the **eserv.config** file. The following parameters are supported.

`balanceTypes`

Syntax:	`balanceTypes = [`*`config`*` ]`
Description:	A list parameter containing identifiers for service providers. For each service provider (ACS customer) configure parameters for the PDSMS balance type.
Type:	Array
Optionality:	Mandatory
Allowed:	
Default:	
Notes:	
Example:	

`ServiceProviderID`

Syntax:	`ServiceProviderID = id`
Description:	The identification number of an ACS customer.
Type:	Integer
Optionality:	Mandatory. At least one ID must be mapped.
Allowed:	
Default:	1
Notes:	
Example:	`ServiceProviderID = 1`

`ThresholdCacheValidityPeriod`

Syntax:	`ThresholdCacheValidityPeriod = minutes`
Description:	The number of minutes between threshold table refreshes from DB.
Type:	Integer
Optionality:	Mandatory
Allowed:	
Default:	10
Notes:	Each threshold table is cached for performance reasons. This period indicates how long each cached table remains valid before being flushed and repopulated from the database.
Example:	`ThresholdCacheValidityPeriod = 10`

`TypeID`

Syntax:	`TypeID = id`
Description:	The PDSMS balance type number for the ACS customer.
Type:	Integer
Optionality:	Mandatory
Allowed:	
Default:	
Notes:	
Example:	`TypeID = 7`

Example

An example of the `ccsPromotionalDestinationSMS` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsPromotionalDestinationSMS = {
    balanceTypes = [
        {
            ServiceProviderID = 1
            TypeID = 7
        }
        {
            ServiceProviderID = 2
            TypeID = 8
        }
        {
            ServiceProviderID = 3
```

```
            TypeID = 5
        }
    ]
    ThresholdCacheValidityPeriod = 10
}
```

ccsRewardsPlugin

Purpose

ccsRewardsPlugin handles the balance changes due to heavy use rewards. For more information about heavy user rewards, see *Recharges* (on page 37).

This plug-in triggers on wallet activated, bucket value changed and bucket expiry events.

Startup

If ccsRewardsPlugin is included in the beVWARS `plugins` array in **eserv.config**,it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsRewardsPlugin.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Parameters

The ccsRewardsPlugin supports the following parameters from the `CCS.ccsRewards` section of **eserv.config**.

`balanceTypes`

Syntax:	`balanceTypes = [config]`
Description:	This section configures which the balance types can be used for rewards for each service provider.
Type:	Parameter array
Optionality:	Mandatory for ccsRewardsPlugin.
Allowed:	
Default:	None
Notes:	You need to add a new service provider in this config file each time one is added in the database.

Example:	`balanceTypes = [{`
	`id = 1`
	`allowed = [ 1 ]`
	`expenditure = 4`
	`notification = [ 1`
	`}`
	`{`
	`id = 2`
	`allowed = [ 5,6 ]`
	`expenditure = 7`
	`}`
	`]`

allowed

Syntax: `allowed = [ <id>,...]`
Description: Lists the balance types that can contribute towards monthly expenditure.
Type: Array
Optionality: Mandatory if expenditure rewards are used.
Allowed:
Default: None
Notes: Must match balance type ids in E2BE database.
This is part of the *balanceTypes* (on page 234) parameter array.
Example: `allowed = [ 1,2,8 ]`

expenditure

Syntax: `expenditure = [ id,... ]`
Description: The balance type for the monthly expenditure.
Type: Array
Optionality: Mandatory if monthly expenditure is used.
Allowed:
Default:
Notes: Must match balance type IDs in E2BE database.
This parameter is part of the *balanceTypes* (on page 234) array.
Example: `expenditure = [ 4 ]`

id

Syntax: `id = id`
Description: The service provider ID for the balance types.
Type: Integer
Optionality: Required
Allowed:
Default:
Notes: Must match service provider ID in E2BE database.
This parameter is part of the *balanceTypes* (on page 234) array.
Example: `id = 1`

notification

Syntax: `notification = [ id,... ]`
Description: Lists the balance types to go in notification short message.
Type: Array
Optionality: Mandatory if notifications are to report any balance types.
Allowed:
Default: None
Notes: Must match balance type ids in E2BE database.
This parameter is part of the *balanceTypes* (on page 234) array.
Example: `notification = [ 1,8 ]`

cacheFlushPeriod

Syntax:	`cacheFlushPeriod = seconds`
Description:	The number of seconds before the reward definition caches are cleared and reloaded.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	600
Default:	
Notes:	
Example:	`cacheFlushPeriod = 600`

cacheValidityTime

Syntax:	`cacheValidityTime = seconds`
Description:	The number of seconds entries are valid for, before a re-read for that reward definition record is required.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	30
Notes:	When ccsRewardsPlugin needs to look up a reward definition, it will check whether the reward definition in the cache is older than this number of seconds. If it is, ccsRewardsPlugin will refresh the cache entry for that reward definition.
Example:	`cacheValidityTime = 30`

cmnPushFiles = [ ]

For the **eserv.config** on the VWS, use the cmnPushFiles configuration to transfer files to the SMS ready for processing by ccsRewardsBatch. Include the `-F` option to detect the file in use.

Note: These directories must match the respective directories set in *writeDirectoryName* (on page 237) and readDirectoryName.

For more information about configuring cmnPushFiles, see *cmnPushFiles* (on page 277).

fileIdleTime

Syntax:	`fileIdleTime = seconds`
Description:	The maximum number of seconds an output file from the ccsRewardsPlugin can be idle before the plug-in will close it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	10
Notes:	
Example:	`fileIdleTime = 30`

filePrefix

Syntax:	`filePrefix = "prefix"`
Description:	The prefix for files: • Written by ccsRewardsPlugin to *writeDirectoryName* (on page 237) • Read by ccsRewardsBatch from readDirectoryName

Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: "ccsRewards"
Notes:
Example: `filePrefix = "ubeprod01-rewards-"`

fileSuffix

Syntax: `fileSuffix = "suffix"`
Description: The suffix for files:
- Written by ccsRewardsPlugin to *writeDirectoryName* (on page 237)
- Read from ccsRewardsBatch from readDirectoryName

Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: ".txt"
Notes:
Example: `filesuffix = ".txt"`

maxLinesInFile

Syntax: `maxLinesInFile = num`
Description: The maximum number of lines in an output file before it is closed.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default: 100
Notes:
Example: `maxLinesInFile = 500`

oracleUserPass

Syntax:
Description: User name and password for connecting to local database (SMF).
Type:
Optionality: This parameter is optional.
Allowed:
Default: "/"
Notes:
Example:

writeDirectoryName

Syntax: `writeDirectoryName = "dir"`
Description: Name of the directory where ccsRewardsPlugin writes its output files.
Type: String
Optionality: Optional (default used if not set).

Allowed:
Default: "/IN/service_packages/CCS/logs/ccsRewardsWrite/"
Notes:
Example: `writeDirectoryName = "/var/logs/Rewards/"`

Example

This text shows an example of the `ccsRewards` section of **eserv.config**.

```
ccsRewards = {
    oracleUserPass = "/"
    fileIdleTime = 10
    maxLinesInFile = 100

    writeDirectoryName = "/IN/service_packages/CCS/logs/ccsRewards/"
    readDirectoryName = "/IN/service_packages/CCS/logs/ccsRewards/"
    filePrefix = "ccsRewards"
    fileSuffix = ".txt"

    cmnPushFiles = [
        "-d", "/IN/service_packages/CCS/logs/ccsRewards/"
        "-r", "/IN/service_packages/CCS/logs/ccsRewards/"
        "-h", "ctelsmp"
        "-p", "2027"
        "-F"
    ]
    balanceTypes = [
        {
            id = 1
            allowed = [ 1 ]
            expenditure = 4
            notification = [ 1 ]
        }
        {
            id = 2
            allowed = [ 5,6 ]
            expenditure = 7
        }
    ]

    cacheFlushPeriod = 600
    cacheValidityPeriod = 30
}
```

Note: This section is also used by ccsRewardsBatch on the SMS and ccsMacroNodes on the SLC.

ccsPMXPlugin

Purpose

ccsPMXPlugin handles the balance changes due to promotions. This plug-in triggers on wallet and balance events, for example:

- Wallet activation
- Wallet expiry
- Balance expiry
- Balance charge
- Balance recharge

- Tracker threshold
- Tracker expiry

This plug-in receives an event and attempts to apply the promotion definitions that match the event type. Matching promotions will be applied providing the conditions configured in the promotion definition are met.

Note: Promotions are configured in the Promotion Manager screen.

Licence

ccsPMXPlugin is only available if the Promotion Manager license has been purchased.

Startup

If ccsPMXPlugin is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsPMXPlugin.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Parameters

The ccsPMXPlugin supports the following parameters from the `CCS.ccsPMXPlugin` section of **eserv.config**.

`cacheValidityTime`

Syntax:	`cacheValidityTime = seconds`
Description:	The length of time in seconds that an entry will be valid for, before the promotion definition record must be reloaded.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	Numerical value
Default:	30
Notes:	When ccsPMXPlugin needs to look up a promotion definition, it will check whether the promotion definition in the cache is older than this number of seconds. If it is, ccsPMXPlugin will refresh the cache entry for that promotion definition.
Example:	`cacheValidityTime = 30`

`ccsBplServiceHandle`

Syntax:	`ccsBplServiceHandle = "service_name"`
Description:	The service name to use when triggering a control plan to recharge third-party balance types.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"CCS_BPL"

Notes:	For the control plan to trigger ACS and the SLEE must be configured with this service name mapped to the CCS service loader.
Example:	`ccsBplServiceHandle = "CCS_BPL"`

`dapInterfaceName`

Syntax:	`dapInterfaceName = "`*`name`*`"`
Description:	The name of the DAP interface running on the VWS
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	A valid DAP interface name
Default:	dapIF
Notes:	
Example:	`dapInterfaceName = "dapIF"`

`rechargeControlPlan`

Syntax:	`rechargeControlPlan = "`*`name`*`"`
Description:	The name of the control plan to use for recharging third-party balance types.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	Either a predefined Promotion Manager control plan or an Open Notifications eRetail control plan.
Default:	"CCS_WebService_Recharge"
Notes:	The Promotion Manager control plan must contain a Voucher Type Recharge node to recharge the third party balance type.
Example:	`rechargeControlPlan = "CCS_WebService_Recharge"`

`rechargeOperationName`

Syntax:	`rechargeOperationName = "`*`name`*`"`
Description:	The name of the OSD operation to use when triggering a control plan to recharge a third-party.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	A valid OSD operation name.
Default:	"applyReward"
Notes:	
Example:	`rechargeOperationName = "applyReward"`

Example

This text shows an example of the `ccsPMXPlugin` section of **eserv.config**.

```
ccsPMXPlugin = {
    cacheValidityTime = 30
    rechargeControlPlan = "CCS_WebService_Recharge"
    ccsBplServiceHandler = "CCS_BPL"
    rechargeOperationName = "applyReward"
    dapInterfaceName = "dapIF"
}
```

ccsVWARSActivation

Purpose

This beVWARS plug-in activates wallets, and optionally credits them with the appropriate balances (from the product type).

Note: If the VWS is defined as a tracking domain only, then only tracking domain balances (fraud and expense balance types) will be updated.

On activation of a wallet (wallet activation event, state change from PreUse to Active), from:

- The product type (CCS_ACCT_TYPE): set the wallet expiry date to the current time + INIT_ACCT_EXPIRY_PERIOD
- CCS_PROMOTION: give the Wallet the promotional amount for the selected balance type and set the expiry date
- The product type (CCS_ACCT_TYPE): set the bucket expiry dates to the current time + EXPIRATION

Note: This can include free SMS buckets.

For more information about wallet states, see *VWS Technical Guide*.

Startup

If ccsVWARSActivation is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsVWARSActivation.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Parameters

The ccsVWARSActivation handler supports the following parameters in the CCS section of **eserv.config**.

`accountBatchCacheValidityPeriod`

Syntax:	`accountBatchCacheValidityPeriod = seconds`
Description:	Time to leave entries in the CCS_ACCT_BATCH cache.
Type:	Integer
Optionality:	Optional.
Allowed:	
Default:	60
Notes:	
Example:	`accountBatchCacheValidityPeriod = 60`

`alwaysOverwriteBucketExpiry`

Syntax:	`alwaysOverwriteBucketExpiry = true|false`
Description:	If true, always set the wallet's buckets' expiry dates, even if these are earlier than the existing bucket's expiry dates.

Type: Boolean
Optionality: Optional.
Allowed: true, false
Default: false
Notes:
Example: `alwaysOverwriteBucketExpiry = false`

`alwaysOverwriteNonExpiringBucketExpiry`

Syntax: `alwaysOverwriteNonExpiringBucketExpiry = true|false`
Description: If the existing bucket never expires, overwrite the expiry date.
Type: Boolean
Optionality: This parameter is optional.
Allowed: true, false
Default: true
Notes:
Example: `alwaysOverwriteNonExpiringBucketExpiry = true`

`alwaysOverwriteNonExpiringWalletExpiry`

Syntax: `alwaysOverwriteNonExpiringWalletExpiry = true|false`
Description: If the existing wallet never expires, overwrite the expiry date.
Type: Boolean
Optionality: Optional.
Allowed: true, false
Default: true
Notes:
Example: `alwaysOverwriteNonExpiringWalletExpiry = true`

`alwaysOverwriteWalletExpiry`

Syntax: `alwaysOverwriteWalletExpiry = true|false`
Description: If true, always set the wallet expiry date, even if this is earlier than the existing wallet expiry date.
Type: Boolean
Optionality: Optional.
Allowed: true, false
Default: false
Notes:
Example: `alwaysOverwriteWalletExpiry = false`

Example

An example of the `ccsVWARSActivation` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsVWARSActivation = {
    accountBatchCacheValidityPeriod = 60
    alwaysOverwriteWalletExpiry = false
    alwaysOverwriteNonExpiringWalletExpiry = true
    alwaysOverwriteBucketExpiry = false
```

```
    alwaysOverwriteNonExpiringBucketExpiry = true
}
```

ccsVWARSAmountHandler

Purpose

beVWARS handler for handling messages relating to rate requests (seconds and named events) and OSA CHAM amounts.

Startup

If ccsVWARSAmountHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

```
handlers = [
    "ccsVWARSAmountHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Configuration

ccsVWARSAmountHandler is configured by the `amountHandler` section of **eserv.config**. This text shows an example of the section.

```
amountHandler = {
    syslogErrors = true|false
}
```

ccsVWARSAmountHandler must also have the appropriate not end actions configured in the beServer section.

```
{type="IARR", action="ACK "}
{type="SARR", action="ACK "}
{type="SARR", action="NACK"}
```

syslogErrors

Syntax:	`syslogErrors = true|false`
Description:	Whether or not to log unspecified wallet errors for IARR and DA messages.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Log the errors to the syslog. false — Do not log the errors to the syslog.
Default:	false
Notes:	The logging of specific wallet errors is not be affected by this parameter.
Example:	`syslogErrors = false`

ccsVWARSExpiry

Purpose

ccsVWARSExpiry is a beVWARS event plug-in which maintains wallet states. This includes:

- Triggering on wallet queries to:
  - Expire PreUse wallets if their subscriber account batch has expired (it also stops actions being taken on PreUse wallets with inactive subscriber account batches)
  - Move wallets from Dormant to Active if they have been used
  - Move wallets from Active to Dormant or Dormant to Terminated if they have not been used for a configurable period of time
  - Remove wallets which have been in a Terminated state for a configurable period of time
  - If expiryAtMidnightTZ is set to true, expire periodic charge buckets
- Triggering on wallet expiry to remove wallets
- Logging wallet removals (this can also be sent to the HLR to update HLR MSISDN records)
- Writing EDRs for most changes (including state changes and removals and bucket removals).

For more information about how ccsVWARSExpiry works with ccsWalletExpiry to manage wallet expiry and removal, see *Subscriber Accounts and Wallet Management* (on page 16).

For more information about subscriber account batches, see *Charging Control Services User's Guide*.

Note: Wallets and buckets can also be expired by the VWS beVWARS plug-in beVWARSExpiry. For more information about beVWARSExpiry, see *VWS Technical Guide*.

Startup

If ccsVWARSExpiry is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsVWARSExpiry.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugin`s array.

Configuration

The ccsVWARSExpiry beVWARS plug-in supports parameters from the notificationPlugin parameter group in the **eserv.config** file on a VWS. It contains parameters arranged in the structure shown below.

Note: This configuration is also used by ccsWalletExpiry on the SMS.

```
ccsVWARSExpiry = {
    expiredPrefix = "prefix"
    expiredDirectory = "dir"
    expiredSuffix = "suffix"
    expiredMaxAge = seconds
    expiryWalletStates = "str[...]"
    produceCDRForWalletExpiredBucket = true|false
    removedDirectory = "dir"
    removedPrefix = "prefix"
    removedSuffix = "suffix"
    removedMaxAge = seconds

    accountBatchCacheValidityPeriod = seconds
    logNotRemoveWallet = true|false
```

```
    removeAtMidnightTZ = "tz"
    cmnPushFiles = [
        "-d", "dir"
        "-r", "dir"
        "-h", "host"
        "-p", "host"
        "-F"
    ]
    deleteEmptyBalances = true|false
}
```

Note: ccsVWARSExpiry also uses the `expireAtMidnightTZ` parameter which is set in the `BE.beVWARSExpiry` section.

Parameters - CCS section

ccsVWARSExpiry supports the following parameters from the `CCS` section of **eserv.config**.

`accountBatchCacheValidityPeriod`

Syntax:	`accountBatchCacheValidityPeriod = seconds`
Description:	The number of seconds an item may stay in the subscriber account batch (CCS_ACCT_BATCH) cache before being re-read from the E2BE database.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60
Notes:	
Example:	`accountBatchCacheValidityPeriod = 120`

`cmnPushFiles = [ ]`

Syntax:	`cmnPushFiles = []`
Description:	For the **eserv.config** on the VWS, use the cmnPushFiles configuration to transfer files to the SMS ready for processing by ccsExpiryMessageLoader.
Type:	Parameter array
Optionality:	Mandatory
Allowed:	
Default:	
Notes:	Include the `-F` option to detect the file in use. See *cmnPushFiles* (on page 277) for all parameters. These directories must match the respective directories set in `generatorFiledir`.
Example:	

createEdrForExpiredValue

Syntax: `createEdrForExpiredValue = "nonzero"|"all"`

Description: Sets whether or not to create an EDR for expired balances with 0 (zero) value. When `createEdrForExpiredValue` is set to:

- "all", ccsVWARSExpiry creates an EDR for all expired balances including those with 0 value
- "nonzero", ccsVWARSExpiry creates an EDR only for expired balances that are greater than 0

Type: String

Optionality: Optional (default used if not set)

Allowed: all, nonzero

Default: nonzero

Notes:

Example: `createEdrForExpiredValue = "all"`

deleteEmptyBalances

Syntax: `deleteEmptyBalances = true|false`

Description: If set, `ccsVWARSExpiry` will delete balances that have both no buckets remaining and the "delete" flag set.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed: true, false

Default: false

Notes:

Example: `deleteEmptyBalances = false`

expiredDirectory

Syntax: `expiredDirectory = "dir"`

Description: Defines the location of files listing wallets moving to terminated state.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "/IN/service_packages/CCS/logs/wallet"

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

Example: `expiredDirectory = "/var/CCS/expiredWallets"`

expiredMaxAge

Syntax: `expiredMaxAge = seconds`

Description: The number of seconds before closing file listing wallets moving to terminated state and creating a new one.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 60

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

Example: `expiredMaxAge = 120`

expiredPrefix

Syntax: `expiredPrefix = "prefix"`

Description: The prefix of files listing wallets moving to terminated state.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "expiredWallet"

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

The filename format is: *expiredPrefix_YYYYMMDDHHMMSSexpiredSuffix*

Example: `expiredPrefix = "produbе01_termWallets"`

expiredSuffix

Syntax: `expiredSuffix = "suffix"`

Description: The suffix of files listing wallets moving to Terminated state.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: .log

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

The filename format is: *expiredPrefix_YYYYMMDDHHMMSSexpiredSuffix*

Example: `expiredSuffix = ".log"`

expireNegativeExpenditureBuckets

Syntax: `expireNegativeExpenditureBuckets = true|false`

Description: If set, ccsVWARSExpiry will expire negative expenditure buckets.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed: true, false

Default: false

Notes:

Example: `expireNegativeExpenditureBuckets = false`

expiryWalletStates

Syntax: `expiryWalletStates = "str[...]"`

Description: Specifies the valid wallet states when ccsVWARSExpiry processes bucket expirations.

Type: String

Optionality: Optional (default used if not set)

Allowed:
- A – Active
- D – Dormant
- F – Frozen
- P – Pre-use
- S – Suspended
- T – Terminated

Default: Active and Dormant

Notes:

Example: `expiryWalletStates = "AD"`

includeExpiredBalanceNames

Syntax: `includeExpiredBalanceNames = true|false`

Description: Whether or not to output the expired balance names in the BALANCE_TYPE_NAMES field in EDRs.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed: true, false

Default: false

Notes:

Example: `includeExpiredBalanceNames = false`

logNotRemoveWallet

Syntax: `logNotRemoveWallet = true|false`

Description: ccsVWARSExpiry plugin will log, and not remove the wallet, so that screen queries will still succeed (they will fail if they have CCS rows but no wallet).

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true — If ccsVWARSExpiry is processing a wallet which has been queried and is in the Terminated state, it will log the wallet's ID to the remove List.

If ccsVWARSExpiry is processing a wallet which has been removed, no action will be taken.

false — If ccsVWARSExpiry is processing a wallet which has been queried and is in the Terminated state, it will:
- Log an EDR detailing the wallet removal
- Remove all the buckets associated with the wallet
- Log EDRs for each bucket which is being removed
- Remove the wallet

If ccsVWARSExpiry is processing a wallet which has been removed, it will log the wallet's ID to the remove List.

Default: true

Notes: If ccsVWARSExpiry does not remove the wallet, ccsWalletExpiry will remove the wallet when it processes the list of wallets to be removed from ccsVWARSExpiry.

Example: `logNotRemoveWallet = false`

produceCDRForWalletExpiredBucket

Syntax: `produceCDRForWalletExpiredBucket = true|false`

Description: Whether or not to produce an EDR for buckets which are expired because they are attached to a wallet which has expired.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

false	The plug-in will not produce an EDR for a bucket which has been expired and has an expiry date in the future or no expire date (as can happen when it is expired as part of a wallet expiry). An EDR will still be produced if the bucket does have an expiry date in the past (so both the bucket and the wallet were due to expire)
true	A bucket expiry EDR will always be produced if the bucket is expired, whether the expiry date is past, present, or future, or it has no expiry date at all.

Default: false

Notes: For more information about when buckets are expired due to their wallet expiring, see *VWS Technical Guide*.

Example: `produceCDRForWalletExpiredBucket = true`

removeAtMidnightTZ

Syntax: `removeAtMidnightTZ = "tz"`

Description: Sets wallets and buckets to be removed at midnight for the time zone specified:

- Midnight GMT (UTC) following the expiry trigger from beVWARSExpiry
- Midnight in the specified timezone after the expiry trigger from beVWARSExpiry
- The time specified by the expiry date

Type: String

Optionality: Optional (default used if not set).

Allowed: The time zone part of the parameter must be typed in a form that the operating system recognizes.

Alternatively you can select a time zone from the operating system's list. To view top-level time zone names, enter `ls /usr/share/lib/zoneinfo` from a shell. To see second-level time zone names, enter `ls /usr/share/lib/zoneinfo TopLevelName/`. For example, to verify that the operating system recognizes a time zone name for DeNoranha, in Brazil, you would enter `ls /usr/share/lib/zoneinfo/Brazil/`. DeNoranha is listed, so the time zone name would be `"Brazil/DeNoranha"`.

Default: Use time specified by the expiry date.

Notes:	The wallet is expired by beVWARSExpiry depending on its configuration. However, will be expired when the wallet is next processed by beVWARSExpiry. The timing of the beVWARSExpiry processing depends on the activity on the VWS. Generally, beGroveller will process the wallet To remove the wallets during the night, the groveller must be set to start after midnight, but before any other access is likely to happen. A list of time zones can be found in the Time Zones appendix of *ACS Technical Guide*.
Example 1:	`removeAtMidnightTZ = "GMT0"`
Example 2:	`removeAtMidnightTZ = "Brazil/DeNoranha"`

removedDirectory

Syntax:	`removedDirectory = "`*`dir`*`"`
Description:	Defines the location of files listing wallets being removed.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"/IN/service_packages/CCS/logs/wallet"
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.
Example:	`removedDirectory = "/var/CCS/removedWallets"`

removedMaxAge

Syntax:	`removedMaxAge =` *`seconds`*
Description:	The number of seconds before closing file listing wallets being removed and creating a new one.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS. Whether ccsVWARSExpiry or ccsWalletExpiry removes the wallet depends on *logNotRemoveWallet* (on page 248).
Example:	`removedMaxAge = 120`

removedPrefix

Syntax:	`removedPrefix = "`*`prefix`*`"`
Description:	The prefix of files listing wallets being removed from the system.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"removedWallet"
Notes:	The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS. Whether ccsVWARSExpiry or ccsWalletExpiry removes the wallet depends on *logNotRemoveWallet* (on page 248). The filename format is: *removedPrefix_YYYYMMDDHHMMSSremovedSuffix*

Example: `removedPrefix = "produbе01_removeWallets"`

`removedSuffix`

Syntax: `removedSuffix = "suffix"`

Description: The suffix of files listing wallets being removed from the system.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: .log

Notes: The file is generated by ccsVWARSExpiry on the VWS and read by ccsWalletExpiry on the SMS.

Whether ccsVWARSExpiry or ccsWalletExpiry removes the wallet depends on *logNotRemoveWallet* (on page 248).

The filename format is: *removedPrefix_YYYYMMDDHHMMSSremovedSuffix*

Example: `removedSuffix = ".log"`

`renewPCAtMidnightTZ`

Syntax: `renewPCAtMidnightTZ = "tz"`

Description: If specified, sets periodic charge balances to expired from midnight (00:00 hrs; the beginning of the day) on the expiry date for the time zone specified.

Type: String

Optionality: Optional (default used if not set)

Allowed: A valid time zone. For more information, see the Time Zones appendix of *ACS Technical Guide*.

Default: Not set

Notes:

Example: `renewPCAtMidnightTZ = "NZ"`

Parameters - BE section

ccsVWARSExpiry supports the following parameters from the `BE` section of **eserv.config**.

`expireAtMidnightTZ`

Syntax: `expireAtMidnightTZ = "tz"`

Description: Sets wallets and buckets to expire at midnight for the time zone specified.

Type: String

Optionality: Optional (default used if not set).

Allowed: The time zone part of the parameter must be typed in a form that the operating system recognizes.

Alternatively you can select a time zone from the operating system's list. To view top-level time zone names, enter `ls /usr/share/lib/zoneinfo` from a shell. To see second-level time zone names enter `ls /usr/share/lib/zoneinfo TopLevelName/`. For example, to verify that the operating system recognizes a time zone name for DeNoranha, in Brazil, you would enter `ls /usr/share/lib/zoneinfo/Brazil/`. DeNoranha is listed, so the time zone name would be `"Brazil/DeNoranha"`.

Default: false (do not modify expiry calculation).

Notes:	A list of time zones can be found in the Time Zones appendix of *ACS Technical Guide*.
Example:	An account is created at 2 p.m. on 5 September 2006 and is set to have a life span of 24 days. If the parameter `expireAtMidnightTZ = "Asia/Vladivostok"` is included, the account will expire on 29 September 2006 at midnight, Vladivostok time. If this parameter is omitted, the account will expire on 29 September 2006 at 2 p.m.

Example

An example of the ccsVWARSExpiry parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsVWARSExpiry = {
    expiredPrefix = "prodube01_expWallet"
    expiredDirectory = "/IN/service_packages/CCS/logs/wallet"
    expiredSuffix = ".log"
    expiredMaxAge = 60
    removedDirectory = "/IN/service_packages/CCS/logs/wallet"
    removedPrefix = "prodube01_rmvWallet"
    removedSuffix = ".log"
    removedMaxAge = 60

    accountBatchCacheValidityPeriod = 60

    logNotRemoveWallet = true

    expiredMsisdnPath="/IN/service_packages/CCS/tmp"
    expiredMsisdnPrefix="prodube01_MSISDNExp"
    expiredMsisdnMaxAge = 120
    removeAtMidnightTZ = "GMT0"
    cmnPushFiles = [
        "-d", "/IN/service_packages/CCS/logs/wallet"
        "-h", "SMF_HOST"
        "-p", "2027"
        "-P", "HOST_NAME"
    ]
}
```

ccsVWARSNamedEventHandler

Purpose

This beVWARS message handler performs the VWS-side processing of messages relating to named events. This includes:

- Returning the desired cost for an event class and event name combination (discounts will be applied to the rates returned)
- Generating named event EDRs

Tariffs are based on the information replicated to the CCS part of the E2BE database.

Named events include GSM notifications, product type swaps, and other discrete billing events. Named events can be performed as either a single-shot or a reserve/commit pair. The type of transaction used will depend on the service's requirement to reverse the charge from the customer based on other events.

Startup

If ccsVWARSNamedEventHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

```
handlers = [
    "ccsVWARSNamedEventHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Configuration

The ccsVWARSNamedEventHandler beVWARS handler supports parameters from the `namedEventHandler` parameter group in the **eserv.config** file on a Voucher and Wallet Server. It contains parameters arranged in the structure shown below.

```
namedEventHandler = {
    maxWalletLockLength = millisecs

    # cascade to use for non promotional Named Events
    CascadeNamesByAcsId = [
        {
        acsCustomerId = ID1
        cascade = "name"
        }
        {
        acsCustomerId = ID2
        cascade = "name"
        }
    ]

    # cascade to use for promotional Named Events

    PromoCascadeNamesByAcsId = [
        {
        acsCustomerId = ID1
        promo_cascade = "promo_name"
        }
        {
        acsCustomerId = ID2
        promo_cascade = "promo_name"
        }
    ]

    reservationPeriod = millisecs
    reservationPeriodTolerance = seconds
    eventCacheAgeSeconds = seconds
    activatePreuseAccount = true|false
    roundingRuleType = "type"
}
```

Parameters

The ccsVWARSNamedEventHandler supports the following parameters in the namedEventHandler section of **eserv.config**.

activatePreuseAccount

Syntax: `activatePreuseAccount = true|false`

Description: When true, activate pre-use wallets for NE and INER requests.

Type: Boolean

Optionality:

Allowed: true, false

Default: true

Notes:

Example: `activatePreuseAccount = true`

acsCustomerId

Syntax: `acsCustomerID = ID`

Description: The ID of the ACS customer.

Type: Integer

Optionality: Required

Allowed: A valid ID for an existing ACS customer.

Default:

Notes:

Example: `acsCustomerID = 12`

CascadeNamesByAcsId

Syntax:

```
CascadeNamesByAcsId = [
        {
        acsCustomerId = ID1
        cascade = "name"
        }
        [{
        acsCustomerId = ID2
        cascade = "name"
        }]
        ]
```

Description: Defines the default balance type cascades for non-promotional named events on a per ACS customer basis.

Type: Array

Optionality:

Allowed:

Default:

Notes:

Example: See *Example* (on page 257) configuration.

cascade

Syntax: `cascade = "name"`

Description: The name of the default balance type cascade for non-promotional named events for the ACS customer specified in *acsCustomerId* (on page 254).

Type: String

Optionality:

Allowed:

Default: "EventCascade"

Notes:

Example: `cascade = "NE Test Cascade""`

eventCacheAgeSeconds

Syntax: `eventCacheAgeSeconds = seconds`

Description: How long to keep named events CCS_EVENT_CLASS, CCS_EVENT_CHARGE, CCS_ACCT_EVENT_CHANGE entries in the cache.

Type: Integer

Optionality:

Allowed:

Default: 600

Notes:

Example: `eventCacheAgeSeconds = 600`

maxWalletLockLength

Syntax: `maxWalletLockLength = millisecs`

Description: How long to lock the wallet for.

Type: Integer

Optionality:

Allowed:

Default: 10000

Notes:

Example: `maxWalletLockLength = 10000`

PromoCascadeNamesByAcsId

Syntax:

```
PromoCascadeNamesByAcsId = [
        {
        acsCustomerId = ID1
        promo_cascade = "name"
        }
        [{
        acsCustomerId = ID2
        promo_cascade = "name"
        }]
        ]
```

Description: Defines the default balance type cascades for promotional named events on a per ACS customer basis.

Type: Array

Optionality:

Allowed:

Default:

Notes:

Example: See *Example* (on page 257) configuration.

promo_cascade

Syntax: `promo_cascade = "name"`

Description: The name of the default balance type cascade for promotional named events for the ACS customer specified in *acsCustomerId* (on page 254).

Type: String
Optionality:
Allowed:
Default: "EventPromoCascade"
Notes:
Example: `promo_cascade = "NE Test Promo Cascade""`

reservationPeriod

Syntax: `reservationPeriod = millisecs`
Description: How long to reserve monies for named events in milliseconds.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 3600
Notes:
Example: `reservationPeriod = 3600`

reservationPeriodTolerance

Syntax: `reservationPeriodTolerance = seconds`
Description: The number of seconds to tolerate a delay for named events reservations before reporting timeout.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default:
Notes:
Example: `reservationPeriodTolerance = 30`

roundingRuleType

Syntax: `roundingRuleType = "type"`
Description: How to round charging list.
Type: String
Optionality: Optional (default used if not set).
Allowed:
- `bankers` – Apply banker's rounding
  (0.000x to 0.499x rounded down to whole integer 0.5 -> 0.999x - round up to whole integer)
- `ceiling` – Apply ceiling rounding
  (0.000x to 0.999x - round up to whole integer)
- `floor` – Apply floor rounding
  (0.000x to 0.999x - round down to whole integer)

Default:
Notes:
Example: `roundingRuleType = "floor"`

Example

An example of the `namedEventHandler` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
namedEventHandler = {
    maxWalletLockLength = 10000
    CascadeNamesByAcsId = [
        {
        acsCustomerId = 12
        cascade = "Cascade1"
        }
        {
        acsCustomerId = 32
        cascade = "Cascade2"
        }
    ]

    PromoCascadeNamesByAcsId = [
        {
        acsCustomerId = 12
        promo_cascade = "Promo Cascade 1"
        }
        {
        acsCustomerId = 32
        promo_cascade = "Promo Cascade 2"
        }
    ]

    reservationPeriod = 3600
    reservationPeriodTolerance = 30
    eventCacheAgeSeconds = 600
    activatePreuseAccount = true
    roundingRuleType = "floor"
}
```

Failure

If ccsVWARSNamedEventHandler fails, interaction with the wallets from the SLC involving updates to named events will fail.

Output

The ccsVWARSNamedEventHandler writes error messages to the system messages file, and also writes additional output to the beVWARS log. For more information about the beVWARS log, see *VWS Technical Guide*.

ccsVWARSPeriodicCharge

Purpose

This beVWARS plug-in handles periodic charge-specific tasks associated with periodic charge bucket changes.

ccsVWARSPeriodicCharge performs these tasks:

- Triggers on bucket expiry event and handles periodic charge logic when the periodic charge expires (that is, when it triggers the next stage in the periodic charge cycle). For more information about the periodic charge life cycle, see *Charging Control Services User's Guide*.

- Triggers on bucket value changed event (set by ccsVWARSWalletHandler when it processes a WU_Req) and handles updating the periodic charge bucket for a new periodic charge state. For subscriptions, creates new balance type and bucket.
- Triggers on wallet state change event or a balance value change event and checks for periodic charges which are in the grace state. For each one it finds it attempts the charge (NE_Req).
  - If successful, all backlogged charges will be applied for the current periodic charge.
  - If one charge fails, the periodic charge will be moved back to the current grace state.
  - If all backlogged charges are successful, move to an active state.

Notes:

- ccsVWARSPeriodicCharge only acts on periodic charge balances and buckets.
- ccsVWARSWalletHandler handles the initial WU_Req messages and bucket updates (except new subscriptions). These updates trigger extra tasks performed by ccsVWARSPeriodicCharge.

For more information about how these tasks fit into the overall periodic charging functionality, see *Periodic Charges* (on page 31).

Startup

If ccsVWARSPeriodicCharge is included in the beVWARS `plugins` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsVWARSPeriodicCharge.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other event plug-ins may also be included in the `plugins` array.

Configuration

The ccsVWARSPeriodicCharge beVWARS event plug-in supports parameters from the `ccsVWARSPeriodicCharge` parameter group in the **eserv.config** file on a Voucher and Wallet Server. It contains parameters arranged in the structure shown below.

```
ccsVWARSPeriodicCharge = {
    retryTimeoutMinutes = mins
    chargeTimeGMTHours = HH
    cacheTimeoutSeconds = seconds
    notificationMidnightTZ = "tz"
    noNotifsInvalidWallet = true|false
    useNonGMTTimezoneOfTriggeringSource = true|false
    alwaysWrite52EDR = true|false
    subscribeExtendsPCExpiryDate = true|false
    keepGraceExpiryInFuture = true|false
    adjustPcVtrBucketExpiry = true
}
```

Parameters

The ccsVWARSPeriodicCharge supports the following parameters in the `ccsVWARSPeriodicCharge` section of **eserv.config**.

`adjustPcVtrBucketExpiry`

Syntax:	`adjustPcVtrBucketExpiry = true|false`
Description:	When set to true, it sets the PC expiry date to bucket expiry date of the PC balance types in case of Voucher Type Recharges.

Type: Boolean
Optionality: Optional
Allowed: true, false
Default: false
Notes: Scenario: When the monthly PC subscribe date is 31st Jan (month end), and later when you perform monthly PC renewal, it works fine and calculates PC bucket expiry date as 29th Feb. However when you perform the PC renewal again, it calculates PC bucket expiry as 29th March instead of 31st March. This flag allows you to calculate the PC bucket expiry to 31st March.
Example: `adjustPcVtrBucketExpiry = true`

alwaysWrite52EDR

Syntax: `alwaysWrte52EDR = true|false`
Description: Whether or not to write a type 52 EDR record for every state change and every expiry date change. When set to false, a type 52 EDR will not be generated if the state remains the same but the expiry date changes.
Type: Boolean
Optionality: Optional (default used if not set).
Allowed:

true	Write type 52 EDR for every state change including expiry date changes
false	Do not write type 52 EDR when the state remains the same but the expiry date changes

Default: true
Notes:
Example: `alwaysWrite52EDR = true`

cacheTimeoutSeconds

Syntax: `cacheTimeoutSeconds = seconds`
Description: The number of seconds to store entries in the beVWARS periodic charge cache.
Type: Integer
Optionality: Optional (default used if not set).
Allowed: Integer, 1-3600.
Default: 300
Notes:
Example: `cacheTimeoutSeconds = 450`

chargeTimeGMTHours

Syntax: `chargeTimeGMTHours = HH`
Description: The time of day (in GMT) that a charge attempt will be made for fixed-date charges.
Type: Integer
Optionality: Optional (default used if not set).
Allowed: Integer, 0-23
Default: 0

Notes: The hours correspond to the hours in a 24 hour clock. For example, specify 10 pm (2200 hours) as 22. Midnight is 0.

This parameter has no affect on the first charge date other than to set the hour. For example, if there is a fixed periodic charge on the 14th day of each month and this parameter specifies an offset for the charge of 12 hours, the first charge will be in the next month, even if the subscriber subscribes in the time lapse between 0:00 and the offset specified by this parameter, or between 0:00 and 12:00 in this case.

For more information about fixed-date configuration, see *Charging Control Services User's Guide*.

Example: `chargeTimeGMTHours = 22`

keepGraceExpiryInFuture

Syntax: `keepGraceExpiryInFuture = true|false`

Description: This parameter dictates how expiry dates will be calculated when there is a forcible movement of Periodic Charges (PCs) from 'Active' to 'Grace' before actual PC expiry.

During the forcible 'Active' to 'Grace' transition, if this parameter is set to true and new calculated PC expiry is in the past, then the PC expiry will be changed to 'current expiry + grace days'.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed: true, false

Default: false

Notes:

Example: `keepGraceExpiryInFuture = true`

notificationMidnightTZ

Syntax: `notificationMidnightTZ = "tz"`

Description: The timezone to use when calculating when a notification should be sent.

Type: String

Optionality: Optional (default used if not set).

Allowed:

Default: "UTC"

Notes: This parameter controls the timezone the notification send time is calculated in. The time is 00:00 by default, but can be specified in the periodic charge configuration on the Wallet Management screen. For more information, see *Charging Control Services User's Guide*.

Example: `notificationMidnightTZ = "GMT"`

noNotifsInvalidWallet

Syntax: `noNotifsInvalidWallet = true | false`

Description: For wallets in an invalid state, specifies whether ccsVWARSPeriodicCharge suppresses all Periodic Charge (PC) notifications or just the Pre-Charge notifications.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

- true – Suppresses all PC notifications
- false – Suppresses only the PreCharge notifications

Default: false

Notes:

Example: `noNotifsInvalidWallet = true`

useNonGMTTimezoneOfTriggeringSource

Syntax: `useNonGMTTimezoneOfTriggeringSource = true|false`

Description: Sets whether to use the timezone defined in the **Timezone** field in the Periodic Charge, When configuration screen in the SMS UI. Set to:

- true – To use the timezone supplied by the source triggering the periodic charge if the supplied timezone is not GMT or UTC.
- false – To use the timezone defined in the periodic charge **Timezone** field or UTC if the periodic charge definition does not specify a time zone.

Regardless of the value of this parameter, the time zone supplied by the triggering source is always used when there are no periodic charge definitions for the balance type being processed.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true, false

Default: false

Notes: When set to `true`, this parameter provides support for periodic charges based on the subscriber's actual timezone. Because there is limited support for, or lack of capability of the various triggering sources, setting this parameter to `true` can lead to inaccurate or erroneous calculations, and inconsistencies in time when notifications are sent, and when periodic charges are applied. Therefore, you are recommended to always set this parameter to `false`.

Example: `useNonGMTTimezoneOfTriggeringSource = false`

retryTimeoutMinutes

Syntax: `retryTimeoutMinutes = mins`

Description: The number of minutes before reattempting a charge after a VWS error.

Type: Integer

Optionality: Optional (default used if not set).

Allowed: 1-1440

Default: 10

Notes:

Example: `retryTimeoutMinutes = 30`

subscribeExtendsPCExpiryDate

Syntax: `subscribeExtendsPCExpiryDate = true|false`

Description: Controls whether a SUBSCRIBE event changes the expiry date of a periodic charge in pre-charge and grace states.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true|false

true - extend the periodic change expiry date

false - do not extend the expiry date

Default: True

Notes:

Example: `subscribeExtendsPCExpiryDate` = true

Example

An example of the `ccsVWARSPeriodicCharge` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsVWARSPeriodicCharge = {
    cacheTimeoutSeconds = 300
    notificationMidnightTZ = "UTC"
    chargeTimeGMTHours = 0
    useNonGMTTimezoneOfTriggeringSource = false
    retryTimeoutSeconds = 10
    keepGraceExpiryInFuture = true
}
```

Failure

If ccsVWARSPeriodicCharge fails, periodic charges will not be processed. When ccsVWARSPeriodicCharge recovers, it will process the failed periodic charges the next time they are queried.

Output

ccsVWARSPeriodicCharge writes:

- Notifications to notification batch file
- Error messages to the system messages file
- Additional output to the beVWARS log

For more information about the beVWARS log, see *VWS Technical Guide*.

ccsVWARSQuota

About the ccsCWARSQuota Plugin

The ccsVWARSQuota plugin sends a notification to the subscriber each time that updates to the subscriber's quota balance type cause a threshold configured for the subscriber's quota balance type to be breached. You configure the quota thresholds for quota balance types by determining the quota value in a profile field, and then by specifying the threshold as a percentage of the quota value.

For more information about configuring quota balance types and thresholds, see the discussion on configuring balance types in NCC *Charging Control Services User's Guide*.

Startup

To enable ccsVWARSQuota to send quota notifications to the subscriber you must include the ccsVWARSQuota plugin in the beVWARS `plugins` array in **eserv.config**. The ccsVWARSQuota plugin is loaded by beVWARS when beVWARS is initialized.

You include the ccsVWARSQuota plugin by using the following sytax:

```
beVWARS = [
    plugins = [
        "ccsVWARSQuota.so"
    ]
]
```

There are no additional configuration parameters for ccsVWARSQuota in the **eserv.config** file, and ccsVARSQuota does not accept any command line parameters.

Note: Other event plug-ins may also be included in the `plugins` array. For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Failure

If ccsVWARSQuota fails, then the quota notifications configured in the Wallet Management screens in the Prepaid Charging UI will not be sent.

ccsVWARSRechargeHandler

Purpose

ccsVWARSRechargeHandler is a beVWARS message handler which handles general wallet recharges (WGR).

Startup

If ccsVWARSRechargeHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
handlers = [
    "ccsVWARSRechargeHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Parameters

The ccsVWARSRechargeHandler supports parameters from the `ccsVWARSUtils` section of **eserv.config**. For more information, see *Parameters* (on page 293).

ccsVWARSReservationHandler

Purpose

This beVWARS message handler performs the VWS-side processing of all messages relating to chargeable call processing including calculating tariffs for CLI-DN combinations. Discounts are applied after the rate is returned. These messages are the reservation messages, and include:

- Initial Reservation (IR)
- Subsequent Reservation (SR)
- Commit Reservation (CR)
- Revoke Reservation (RR)

Startup

If ccsVWARSReservationHandler is included in the beVWARS `handler` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
handlers = [
    "ccsVWARSReservationHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Parameters

The ccsVWARSReservationHandler supports the following parameters in the reservationHandler section of **eserv.config**.

`addDisplaySpendRatio`

Syntax:	`addDisplaySpendRatio = true|false`
Description:	Enable if display spend ratio is required in the EDR.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	• true - Add spend ratio in the EDR. • false - Do not add spend ratio in the EDR.
Default:	false
Notes:	Setting this parameter to true adds a comma delimited (per balance) spend ratio in the EDR. e.g. DISPLAY_SPEND_RATIO=1.0,2.0,1.0 etc
Example:	addDisplaySpendRatio = false

`addGeoSetID`

Syntax:	`addGeoSetID = true|false`
Description:	Log the geo set entry IDs for CLI and DN into EDR.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	
Default:	false
Notes:	
Example:	`addGeoSetID = true`

`alwaysContributeToXBTDTimeBalance`

Syntax:	`alwaysContributeToXBTDTimeBalance = <true|false>`	
Description:	Indicates how to do handle cross balance duration.	
Type:	Boolean	
Optionality:	Optional	
Allowed:	true	Always debit the duration of the current rate from the **Cross** balance type **Time** balance (if applicable in the current Cross balance type cascade) regardless of whether a wallet discount is being applied to the resulting cost of this rate.
	false	
Default:	false	
Notes:		
Example:	`alwaysContributeToXBTDTimeBalance = false`	

createEDRForMidSessionCommit

Syntax: `createEDRForMidSessionCommit = <true|false>`

Description: Flag to generate a partial EDR for each mid-session commit.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:
- true - generate partial EDR, or
- false - do not generate any partial EDRs

Default: false

Notes:

Example: `createEDRForMidSessionCommit = false`

discountData

Syntax: `discountData = true|false`

Description: Whether or not to discount charges on data balances.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Apply discounts.
false	Do not apply discounts.

Default: true

Notes: For example, if discountData is set to true and you have 40 free data units and a discount of 50%, you will actually get 80 data units of call time.

If discountData is set to false, you will get 40 free data units regardless of applicable discounts.

Example: `discountData = true`

discountRuleType

Syntax: `discountRuleType = "<rule>"`

Description: How to factor service discounts from the IR_Req, SR_Req or CR_Req into the discounts to be applied from the rating and the wallet.

Type: String

Optionality: Optional (default used if not set).

Allowed:	ServiceOverride	override service discounts
	s*r*w	compound service, rating and wallet discounts
	s+r+w	cumulate service, rating and wallet discounts
	s+r*w	cumulate service and rating discounts then compound the result to the wallet discount
	s*r+w	compound service and rating discounts then cumulate the result to the wallet discount
	s+w*r	cumulate service and wallet discounts then compound the result to the rating discount
	s*w+r	compound service and wallet discounts then cumulate the result to the rating discount
	r+w*s	cumulate rating and wallet discounts then compound the result to the service discount
	r*w+s	compound rating and wallet discounts then cumulate the result to the service discount
Default:	s*w*r	
Notes:	s = service. The incoming discounts from the SLC as specified in the IR_Req, SR_Req and CR_Req messages.	
	r = rating. Holiday or weekly discounts that may be applicable during the call.	
	w = wallet. The discounts that are based on specific 'Cross Balance Type Discount' wallet balances being present when the call charge is being calculated.	
Example:	`discountRuleType = "s*r*w"`	

`discountTime`

Syntax:	`discountTime = <true|false>`	
Description:	Whether or not to discount charges on time balances.	
Type:	Boolean	
Optionality:	Optional (default used if not set).	
Allowed:	true	Apply discounts.
	false	Don't apply discounts.
Default:	false	
Notes:	For example, if discountTime is set to true and you have 40 free minutes and a discount of 50%, you will actually get 80 minutes of call time.	
	If discountTime is set to false, you will get 40 free minutes regardless of applicable discounts.	
Example:	`discountTime = true`	

`greedyReservationLengthLimit`

Syntax:	`greedyReservationLengthLimit = <secs>`
Description:	The number of seconds reservation of funds should aim to be.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	

Default: 60

Notes: If this number of seconds cannot be reserved, the wallet is treated as if it has a Maximum Concurrent Accesses of 1. For more information about Maximum Concurrent Accesses settings, see *Charging Control Services User's Guide*.

This parameter does not affect charging for named events.

Example:

maxReservationLength

Syntax: `maxReservationLength = <secs>`

Description: The number of seconds to attempt to reserve for an IR or SR.

Type: Integer64

Optionality: Optional (default used if not set).

Allowed:

Default: 3600

Notes: This is what will be reserved if the wallet has infinite funds.

Example:

reservationLengthTolerance

Syntax: `reservationLengthTolerance = <secs>`

Description: The number of seconds the reservation length should exceed the length of time which can be paid for out of the funds available to the wallet.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 350

Notes: This does not give free call time but allows the application of a CR or SR to be delayed slightly.

Example: `reservationLengthTolerance = 350`

showCostsEDRScaledByDisplaySpendRatio

Syntax: `showCostsEDRScaledByDisplaySpendRatio = true|false`

Description: Whether to show COSTS scaled by the display spend ratio in EDR.

Type: Boolean

Optionality:

Allowed: true - Show COSTS scaled by the display spend ratio.

false - Do not show COSTS scaled by the display spend ratio.

Default: false

Notes: COSTS added to the EDR remove the scaling due to the display spend ratio by default. In order to see the COSTS scaled by the display spend ratio, set showCostsEDRScaledByDisplaySpendRatio to true.

Example: `showCostsEDRScaledByDisplaySpendRatio = false`

suppressEDRRatingDetails

Syntax: `suppressEDRRatingDetails = <true|false>`

Description: Whether to suppress some rating fields in the EDRs written for midcall rating change (FMC) and/or multi tariff rating calls.

Single tariff calls are not affected by this parameter.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Suppressing the fields listed above for FMC and/or MTR calls.
false	Use normal approach to writing fields.

Default: true

Notes: The suppressed fields are:

- RATES
- LENGTHS
- MAX_CHARGE
- DISCOUNTS
- CASCADE_ID
- CBTD_DISCOUNTS, and
- CBTD_CASCADE_ID.

For more information about these EDR fields, see *EDR Reference Guide*.

Example:

syslogErrors

Syntax: `syslogErrors = <true|false>`

Description: Whether or not to log some NACKs to the syslog.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Log all NACKs except MaxConcurrentExceeded, InsufficientFunds, and WalletDisabled to the syslog.
false	Do not log any NACKs to the syslog.

Default: false

Notes: These errors include some detail about why the action failed.

Example:

useWorstCaseBalances

Syntax: `useWorstCaseBalances = <true|false>`

Description: For new calls during initial reservations, whether to use worst case balances only or all available balances.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Use worst case balances only.
false	Use all available balances.

Default: true

Notes: Worst case balances are those which are having a start date and an expiry date. Balances having expiry date, without any start date are considered as worst case balance, only if it is not going to expire within the call start time and a calculated expected call finish time.

Example: `useWorstCaseBalances = false`

zeroLengthFreeCalls

Syntax: `zeroLengthFreeCalls = {}`

Description: How successful, free, zero-length calls should be handled. For example, where the caller hangs up before the call is answered.

Type: Array

Optionality: Optional (not used if not set).

Allowed:

Default:

Notes: These settings can be used to reduce the amount of resources used for successfully placed free calls, which are unanswered.

Example:

updateLastUseDate

Syntax: `updateLastUseDate = <true|false>`

Description: Whether successful, free, zero-length calls should change the wallet's last use date in the database. For example, where the caller hangs up before the call is answered.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Update the wallet's last use date in the database
false	Do not update the wallet's last use date.

Default: true

Notes: This settings can be used to reduce the amount of resources used for successfully placed free calls, which are unanswered.

For more information about Last Use Date, see *Charging Control Services User's Guide*.

Example:

writeCDR

Syntax: `updateLastUseDate = <true|false>`

Description: Whether successful, free, zero-length calls should generate an EDR. For example, where the caller hangs up before the call is answered.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Write an EDR for the call.
false	Do not write an EDR for the call.

Default: true

Notes: This settings can be used to reduce the amount of resources used for successfully placed free calls, which are unanswered.

For more information about EDRs, see *EDR Reference Guide*.

Example:

Example

An example of the reservationHandler parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
reservationHandler = {
    syslogErrors = false
    maxReservationLength = 3600
    reservationLengthTolerance = 30
    greedyReservationLengthLimit = 60
    discountRuleType = "s*w*r"
    alwaysContributeToXBTDTimeBalance = false
    suppressEDRRatingDetails = true
    discountTime = false
    discountData = true
    addGeoSetID = true
    createEDRForMidSessionCommit = false
    addDisplaySpendRatio = false
    showCostsEDRScaledByDisplaySpendRatio = false
    useWorstCaseBalances = false
    zeroLengthFreeCalls = {
        updateLastUseDate = true
        writeCDR = true
    }
}
```

Failure

If ccsVWARSReservationHandler fails, interaction with the subscriber accounts from the SLC involving call charging will fail.

Output

The ccsVWARSReservationHandler writes error messages to the system messages file, and also writes additional output to:

```
/IN/service_packages/E2BE/tmp/beVWARS.log
```

ccsVWARSVoucherHandler

Purpose

This beVWARS message handler performs the Voucher and Wallet Server side processing of messages directly relating to vouchers. This includes voucher reservation/commit, alteration and deletion. It does not perform the wallet recharge; this is done by the *ccsVWARSWalletHandler* (on page 274). The message handler only controls the Voucher and Wallet Server side of the CCS voucher tables, not the main body of data about vouchers that is replicated from the SMS.

This handler validates incoming voucher reserve (for example, scratch or redeem) requests, and refers to the replicated CCS voucher tables for all information except the current redeemed/unredeemed state of the voucher.

It is important to remember that the BE_VOUCHER record will in all probability not exist unless the voucher has had a previous successful (or almost successful) redeem performed upon it. This state is hidden from the client process, a non-existent BE_VOUCHER record is proof that the voucher has not been redeemed.

Startup

If ccsVWARSVoucherHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
handlers = [
    "ccsVWARSVoucherHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Parameters

The ccsVWARSVoucherHandler supports the following parameters in the `beVWARS` section of **eserv.config** .

Note: It also required the `BE.serverId` parameter. For more information about setting `serverId`, see *VWS Technical Guide*.

badPinExpiryHours

Syntax:	`badPinExpiryHours = hours`
Description:	The number of hours before the bucket storing the bad PIN expires.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	negative integer Does not expire positive integer Number of hours before expiry
Default:	24
Notes:	
Example:	`badPinExpiryHours = 48`

clearConsecutivePin

Syntax:	`clearConsecutivePin = Boolean`
Description:	If true, then a successful voucher recharge will set the number of consecutive bad pin attempts for an account to zero.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	
Default:	false
Notes:	
Example:	`clearConsecutivePin = true`

consecutiveBadPinExpiryHours

Syntax:	`consecutiveBadPinExpiryHours = hours`
Description:	The number of hours before the bucket storing the consecutive bad PIN expires.
Type:	Integer
Optionality:	Optional (default used if not set)

Allowed:	negative integer	Does not expire
	positive integer	Number of hours before expiry
Default:	24	
Notes:		
Example:	`consecutiveBadPinExpiryHours = 48`	

createRechargeCDRInactiveAccount

Syntax:	`createRechargeCDRInactiveAccount = true|false`
Description:	When true, failed voucher recharges generate an EDR.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true, false
Default:	true
Notes:	
Example:	`createRechargeCDRInactiveAccount = true`

dailyBadPinExpiryHours

Syntax:	`dailyBadPinExpiryHours = hours`
Description:	The number of hours before the bucket storing the daily bad PIN expires.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	negative integer Does not expire positive integer Number of hours before expiry
Default:	24
Notes:	
Example:	`dailyBadPinExpiryHours = 48`

monthlyBadPinExpiryHours

Syntax:	`monthlyBadPinExpiryHours = hours`
Description:	The number of hours before the bucket storing the monthly bad PIN expires.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	negative integer Does not expire positive integer Number of hours before expiry
Default:	744
Notes:	
Example:	`monthlyBadPinExpiryHours = 744`

weeklyBadPinExpiryHours

Syntax:	`weeklyBadPinExpiryHours = hours`
Description:	The number of hours before the bucket storing the weekly bad PIN expires.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	negative integer Does not expire positive integer Number of hours before expiry

Default: 744
Notes:
Example: `weeklyBadPinExpiryHours = 744`

replicationInterface

Syntax: `replicationInterface = "if"`
Description: The handle of the SLEE replication interface.
Type: String
Optionality: Optional (default used if not set)
Allowed: Must match the Interface name in **SLEE.cfg**.
Default: "replicationIF"
Notes: For more information about **SLEE.cfg**, see *SLEE Technical Guide*.
Example: `replicationInterface = "replicationIF"`

requireBonusRow

Syntax: `requireBonusRow = true|false`
Description: When true, vouchers will fail if there is no entry in CCS_BONUS_VALUES.
Type: Boolean
Optionality: Optional (default used if not set)
Allowed: true, false
Default: true
Notes:
Example: `requireBonusRow = true`

updateLastUseVoucherRecharge

Syntax: `updateLastUseVoucherRecharge = true|false`
Description: When true, voucher recharges update the 'last use date' field.
Type: Boolean
Optionality: Optional (default used if not set)
Allowed: true, false
Default: true
Notes:
Example: `updateLastUseVoucherRecharge = true`

vomsInstalled

Syntax: `vomsInstalled = true|false`
Description: Define if you are using:
- Voucher Manager-type bad PIN balances (true)
- Just a single, VWS bad PIN (false)

Type: Boolean
Optionality: Optional (default used if not set)
Allowed: true, false
Default: false

Notes:

Example: `vomsInstalled = true`

Example

An example of the `voucherHandler` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
voucherHandler = {
    requireBonusRow = true
    updateLastUseVoucherRecharge = true
    createRechargeCDRInactiveAccount = true
    badPinExpiryHours = 24
    dailyBadPinExpiryHours = 24
    monthlyBadPinExpiryHours = 744
    consecutiveBadPinExpiryHours = -1

    vomsInstalled = true
    replicationInterface = "replicationIF"
}
```

Failure

If ccsVWARSVoucherHandler fails, interaction with the wallets from the SLC involving vouchers will fail.

Output

The ccsVWARSVoucherHandler writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/E2BE/tmp/beVWARS.log**.

ccsVWARSWalletHandler

Purpose

This beVWARS message handler performs the VWS side processing of all messages relating directly to wallets. This includes:

- Wallet information (WI) - responds with wallet information
- Wallet create (WC) - creates new wallets
- Wallet update (WU) - updates wallets and possibly adds reload bonuses and writes an EDR.
- Wallet delete (WD) - deletes existing wallets and corresponding buckets
- Bad PIN updates (BPIN) - updates bad PIN balance if the wallet has one.

EDRs are produced for all Wallet updates (create/modify/delete/recharge) with the details of the change.

Note: ccsVWARSWalletHandler only performs some updates for periodic charge balances and buckets. For more information about how ccsVWARSWalletHandler handles WU_Req messages which relate to periodic charges, see Processing periodic charge subscription changes.

For more information about wallet messages, see *VWS Technical Guide*.

Startup

If ccsVWARSWalletHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
handlers = [
    "ccsVWARSWalletHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

Configuration

The ccsVWARSWalletHandler library accepts the following configuration parameter for the ccsWalletUpdateHandler plug-in:

```
walletUpdateHandler = {
    createEmptyBuckets = true|false
    deleteEmptyBuckets = true|false
    maxReservationsPerSLEEMessage = 5
}
```

createEmptyBuckets

Syntax: `createEmptyBuckets = true | false`

Description: Specifies whether ccsVWARSWalletHandler creates empty buckets for subscribers that are added through PI commands or the User Interface (UI).

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:
- true – Empty buckets are created
- false – No empty buckets are created

Default: true

Notes:

Example: `createEmptyBuckets = false`

deleteEmptyBuckets

Syntax: `deleteEmptyBuckets = true|false`

Description: Controls whether beServer deletes empty buckets or whether it is done by beVWARSExpiry (and controlled by its configuration).

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Empty buckets will be removed by the beServer.
false	Empty buckets will be removed by the beVWARSExpiry plug-in.

Default: true

Notes: For more information about beServer and beVWARSExpiry, see *VWS Technical Guide*.

Example: `deleteEmptyBuckets = false`

maxReservationsPerSLEEMessage

Syntax: `maxReservationsPerSLEEMessage = Int`

Description: Specifies the maximum number of reservations returned by the VWS when querying for wallet reservation details.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default:	5
Notes:	Do not set this parameter higher than the maximum SLEE event size (MAXEVENTS). See About Configuring MAXEVENTS. Otherwise, the CCSRM1=QRY PI command time outs, and the VWS generates a no free events error.
Example:	`maxReservationsPerSLEEMessage = 3`

Failure

If ccsVWARSWalletHandler fails, interaction with the wallets from the SLC will fail.

Output

The ccsVWARSWalletHandler writes error messages to the system messages file.

ccsVWARSPolicyHandler

Purpose

This beVWARS policy handler creates sessions in the VWS with triggers for balance change through the counter boundaries. When triggered, notifies ACS of the change via beServiceTrigger and osdInterface, triggering the SNR control plan.

Startup

If ccsVWARSPolicyHandler is included in the beVWARS `handlers` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
handlers = [
    "ccsVWARSPolicyHandler.so"
]
```

For more information about the beVWARS `handlers` section, see *handlers* (on page 211).

Note: Other handlers may also be included in the `handlers` array.

ccsWLCPlugin

Purpose

ccsWLCPlugin is a beVWARS plug-in that handles wallet life cycle periods. It is triggered by a wallet query event and provides the following services:

- Processes wallet life cycle plan and periods transitions
- Triggers on entry and on backout control plans
- Triggers on entry and on backout notifications

Startup

If ccsWLCPlugin is included in the beVWARS `plugin` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsWLCPlugin.so"
```

```
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other plug-ins may also be included in the `plugins` array.

ccsPolicyPlugin

Purpose

This policy plug-in will perform it's usual loop through the wallet balances to determine the affected counters and then loop through all the qualifying sessions, at the end we have a SySessionPayloadMap per session id containing all the qualifying policy counters.

Startup

If ccsPolicyPlugin is included in the beVWARS `plugin` array in **eserv.config**, it is loaded by beVWARS when beVWARS is initialized.

It is included in the following lines:

```
plugins = [
    "ccsPolicyPlugin.so"
]
```

For more information about the beVWARS `plugins` section, see *plugins* (on page 212).

Note: Other plug-ins may also be included in the `plugins` array.

cmnPushFiles

Purpose

cmnPushFiles is responsible for pushing files to other machines including, but not limited to, log files to the SMS.

Warning: You must install the xinetd daemon as a prerequisite to running cmnPushFiles. You install the daemon by entering the following command:

```
yum install xinetd
```

Startup

Each instance of the cmnPushFiles daemon should be started with a separate entry in the inittab of the machine where it will run. It runs under the control of inetd.

Before adding an entry to the inittab, you must decide the following:

1. User you wish to have cmnReceiveFiles write incoming files as (for example, ccs_oper)
2. File names you wish to transfer (for example, file names starting with "ccsCDR")
3. Directories on each host you want to transfer files between (for example,BE/logs/CDR and CCS/logs/CDR)
4. Host name of the receiving side of the connection (for example, hp3)
5. Port number the two programs will use to communicate (for example, 2027)

Receiving machine

You must also ensure a matching cmnReceiveFiles is available on the destination machine.

On the receiving machine, add an entry to **/etc/services** like this:

```
ccsoperFiles    2027/tcp
```

and to **/etc/inetd.conf**, add an entry like this:

```
ccsoperFiles stream tcp nowait root /IN/service_packages/CCS/bin/cmnReceiveFiles
cmnReceiveFiles -u ccs_oper.
```

Parameters

cmnPushFiles supports the following parameters.

`-a`

Syntax:
Description: How old transferred files must be before they are removed.
Type:
Optionality:
Allowed:
Default: Never clean
Notes: This parameter is only relevant if the `-o` parameter is specified.
Example:

`-C`

Syntax:
Description: Clean up period.
Type:
Optionality: Optional
Allowed:
Default: 1800
Notes: In seconds
Example:

`-d`

Syntax:
Description: Scan Directory. The directory to search for files to transfer to the receiving side. cmnPushFiles will only transfer those files matching a pattern. See `-P`.
Type:
Optionality:
Allowed:
Default:
Notes:
Example:

`-f`

Syntax:
Description: Retry directory.
Type:
Optionality: Optional.
Allowed:

Default: none
Notes:
Example:

`-F`

Syntax:
Description: Use fuser to not move files in use.
Type:
Optionality: Optional.
Allowed:
Default: Do not use
Notes:
Example:

`-h`

The host name of the cmnReceiveFiles listener.

`-M`

Syntax:
Description: Maximum retry period.
Type:
Optionality: Optional.
Allowed:
Default: 900
Notes:
Example:

`-o`

Syntax:
Description: Transferred directory. What to do with files that have been transferred.
Type:
Optionality: Optional.
Allowed:
Default: File deleted
Notes:
Example:

`-p`

Syntax:
Description: The port number of the cmnReceiveFiles listener.
Type:
Optionality: This parameter is optional.
Allowed:
Default: 2027
Notes:

Example:

`-P`

Syntax:
Description: Match Pattern. Specify a filename prefix that must be matched in order to qualify a file for transfer to the remote side.
Type:
Optionality: This parameter is optional.
Allowed:
Default:
Notes:
Example: `-P ccsCDR` will cause all files matching `ccsCDR*` in the source directory to be transferred.

`-R`

Syntax:
Description: Initial retry period.
Type:
Optionality: Optional
Allowed:
Default: 15
Notes: In seconds.
Example:

`-r`

Remote directory prefix.

- If the `-r` parameter is omitted, files will be written to the target machine using the path used by the source machine.
- If the `-r` parameter is included, the remote directory prefix is added to the front of all matching file names in the source directory.
- If the `-d` parameter is used and if it specifies a relative directory (one that starts with a `/`), the `-r` parameter must be specified. Otherwise, this parameter is optional.

`-s`

Syntax:
Description: The re-scan interval. After cmnPushFiles has scanned its input directory and found no files to transfer, it goes to sleep for a configurable interval. To change this interval, specify the number of seconds to sleep after the `-s`.
Type:
Optionality: Optional
Allowed:
Default: 15
Notes: In seconds
Example:

`-S`

Syntax:
Description: File suffix.
Type:
Optionality: Optional
Allowed:
Default:
Notes:
Example:

`-t`

Syntax: `-t int`
Description: Throttle. Controls the maximum transmission speed the application will use when transferring data.
Type: Integer
Optionality: Optional.
Allowed:
Default:
Notes: Specify the number of bits per second to use after the option.
Example:

`-T`

Syntax:
Description: Tree move. Recursively moves the directory tree.
Type:
Optionality: Optional
Allowed:
Default: off
Notes:
Example:

`-x`

No host name prefix. By default, cmnPushFiles adds the sending host name to file names sent to the receiving side using the convention: *hostName_fileName*. To prevent the host name being added, use the –x switch. This parameter is optional.

Failure

If cmnPushFiles fails, files will collect in the input directory. When the process starts up again, the unprocessed files will be processed.

If cmnPushFiles fails to copy a file to the remote location, it will move the files into a failed directory.

Output

cmnPushFiles will transfer files to the configured target machine and will move the local files to a completed transfer directory.

The cmnPushFiles writes error messages to the system messages file.

libccsCommon

Purpose

libccsCommon provides common functions to various CCS processes.

Startup

libccsCommon is used by a number of CCS processes. No startup configuration is required for this library to be used.

Configuration

The libccsCommon library supports parameters from the `common` parameter group in the **eserv.config** file on all machines. It contains parameters arranged in the structure shown below. The value types for each parameter are displayed as placeholders.

```
common = {
      # how long entries should remain in the various caches
       balanceTypeCascadeIdCacheAgeSeconds = seconds
       balanceTypeCascadeCacheAgeSeconds = seconds
       balanceTypeUnitCacheAgeSeconds = seconds
       balanceTypeIdCacheAgeSeconds = seconds
       defaultBalanceTypeCacheAgeSeconds = seconds
       systemCurrencyBalanceUnitCacheAgeSeconds = seconds
       accountCacheAgeSeconds = seconds
       accountTypeBestPeriodsCacheAgeSeconds = seconds
       tariffPlanIdCacheAgeSeconds = seconds
       walletTypeCacheAgeSeconds = seconds
       profileDetailsSubtagsCacheAgeSeconds = seconds
       domainsCacheAgeSeconds = seconds
       lowCreditDapDisableCacheAgeSeconds = seconds
       ccsAcctReferenceCacheAgeSeconds = seconds
       productTypeIdCacheAgeSeconds = seconds
       creditCardCacheAgeSeconds = seconds
       creditCardCacheRepIntervalSeconds = seconds

      #the following should go into eserv.config.be
       acsCustIdAgeSeconds = seconds
       ccsBalanceExpiryRoundUp = boolean

      # If set, the timezone in which to set the expiry date of an expenditure
      # balance to the next midnight.
      # default: GMT
      # expenditureBalanceMidnightExpiryTZ = "timezone"
      # mode for CB10 HRN validation.
      # true = force HRN to be validated against seed
      # false = no validation of HRN against seed
      # default: true
       authCB10ValidateSeed = boolean

      # Name of xmlIF used when sending low credit DAP notifications (default xmlIF)
       xmlInterfaceName = "interface name"
      # The length of time in seconds between syslog messages about
      # not being able to send to a SLEE Interface. (BE only)
      # Default 60 (seconds)
       rateLimitIFSendErrors = seconds

}
```

Parameters

libccsCommon library supports these parameters in the common section of eserv.config.

`accountCacheAgeSeconds`

Syntax:	`accountCacheAgeSeconds = seconds`
Description:	The maximum number of seconds that account data remains cached.
Type:	Integer
Optionality:	
Allowed:	
Default:	600
Notes:	
Example:	`accountCacheAgeSeconds = 600`

`accountNumberLength`

Syntax:	`accountNumberLength = len`
Description:	The number of digits in a card account number generated by ccsAccount tool.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	10
Notes:	If you set accountNumberLength to 0 (zero), the account numbers that the ccsAccount tool generates can be any length.
Example:	`accountNumberLength = 10`

`accountTypeBestPeriodsCacheAgeSeconds`

Syntax:	`accountTypeBestPeriodsCacheAgeSeconds = seconds`
Description:	The value specified in the **accountTypeBestPeriodsCacheAgeSeconds** parameter indicate the maximum validity time for data stored in the **accountTypeBestPeriodcache.** The **accountTypeBestPeriodcache** caches the best (that is longest) value of each of the four items listed below from the `CCS_ACCT_TYPE` table, for each wallet. That is, for each item, it caches the longest value across all account types using the wallet. • **ACTIVE_DORMANT** - Number of days an account in the active state can be inactive before it is deemed dormant. • **DORMANT_TERMINATED** - Number of days an account in the dormant state can be inactive before it is deemed terminated. • **PRE_USE_EXPIRY** - Number of days an account in the pre-use state can be inactive before it is deemed terminated. • **TERMINATED_REMOVED** - Number of days an account in the terminated state can be inactive before it is removed from the database. Each row in the `CCS_ACCT_TYPE` table represents a product type. It is possible to have a wallet which is shared by more than one account, each of which has a different product type.
Type:	Integer
Optionality:	
Allowed:	

Default: 600
Notes:
Example: `accountTypeBestPeriodsCacheAgeSeconds = 600`

authCB10ValidateSeed

Syntax: `authCB10ValidateSeed = true|false`
Description: The mode for CB10 HRN validation.
Type: Boolean
Optionality: Optional (default used if not set).
Allowed:
- **true** Force incoming HRN to be validated against originating seed (voucher number).
- **false** Do not validate of HRN against originating seed.

Default: true
Notes: This parameter is only needed if the system is using *ccsCB10HRNSHA* (on page 108) or *ccsCB10HRNAES* (on page 108).

This setting may be useful where:
- Vouchers have been imported from another system and the original voucher number seeds are not available
- Validation performance gains are needed.

Example: `authCB10ValidateSeed = false`

balanceTypeCascadeCacheAgeSeconds

Syntax: `balanceTypeCascadeCacheAgeSeconds = seconds`
Description: The maximum number of seconds that data of the cascade balance type remains cached.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 600
Notes:
Example: `balanceTypeCascadeCacheAgeSeconds = 600`

balanceTypeCascadeIdCacheAgeSeconds

Syntax: `balanceTypeCascadeIdCacheAgeSeconds = seconds`
Description: The maximum number of seconds that data of the cascade identifier balance type remains cached.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 600
Notes:
Example: `balanceTypeCascadeIdCacheAgeSeconds = 600`

detailedCascadeCacheAgeSeconds

Syntax: `detailedCascadeCacheAgeSeconds = seconds`

Description: The maximum number of seconds that data of the cascade detailed balance type remains cached.

Type: Integer

Optionality:

Allowed:

Default: 600

Notes:

Example: `detailedCascadeCacheAgeSeconds = 600`

balanceTypeIdCacheAgeSeconds

Syntax: `balanceTypeIdCacheAgeSeconds = seconds`

Description: The maximum number of seconds that data of the identifier balance type remains cached.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600

Notes:

Example: `balanceTypeIdCacheAgeSeconds = 600`

balanceTypeUnitCacheAgeSeconds

Syntax: `balanceTypeUnitCacheAgeSeconds = seconds`

Description: The maximum number of seconds that data of the unit balance type remains cached.

Type: Integer

Optionality:

Allowed:

Default: 600

Notes:

Example: `balanceTypeUnitCacheAgeSeconds = 600`

balanceUnitTypeCacheAgeSeconds

Syntax: `balanceUnitTypeCacheAgeSeconds = seconds`

Description: The number of seconds before the balance unit cache is refreshed from the database.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600

Notes: The balance unit type configuration is used by the rating engine for determining how a balance type should be treated, (that is, is is cash, time or other). This configuration is based on the name of the unit.

Example: `balanceUnitTypeCacheAgeSeconds = 600`

ccsAcctReferenceCacheAgeSeconds

Syntax: `ccsAcctReferenceCacheAgeSeconds = seconds`

Description: The maximum number of seconds data from the `CCS_ACCT_REEFERENCE` table is cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `ccsAcctReferenceCacheAgeSeconds = 600`

ccsBalanceExpiryRoundUp

Syntax: `ccsBalanceExpiryRoundUp = boolean`

Description: Used when calculating the expiry time of the balance in days, to present to the customer.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true Round up when calculating expiry time.

false Do not round up when calculating expiry time.

Default: false

Notes: Sample scenarios:

- If the balance expires 3.25 days from now, the expiry time will always be given as 3 days.
- If the balance expires 3.75 days from now, and this parameter is **true**, it will be given as 4 days.
- If the balance expires 3.25 days from now, and this parameter is **true**, it will be given as 3 days.

Example `ccsBalanceExpiryRoundUp = true`

ccsBonusTypeAgeSeconds

Syntax: `ccsBonusTypeAgeSeconds = seconds`

Description: The number of seconds before refreshing the bonus type cache from the E2BE database.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600

Notes: This cache holds the balance type ID to apply the bonus to for a given bonus.

Example: `ccsBonusTypeAgeSeconds = 600`

ccsBonusValuesAgeSeconds

Syntax: `ccsBonusValuesAgeSeconds = seconds`

Description: The number of seconds before refreshing the bonus value cache from the E2BE database.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600
Notes: This cache holds the range of valid values and the bonus percentage to give for a given bonus.
Example: `ccsBonusValuesAgeSeconds = 600`

ccsWlcAgeSeconds

Syntax: `ccsWlcAgeSeconds = seconds`
Description: The maximum number of seconds that wallet life cycle data remains in the CCS common cache.
Type: Integer
Optionality:
Allowed:
Default: 600
Notes:
Example: `ccsWlcAgeSeconds = 600`

ccsRewardTranslationAgeSeconds

Syntax: `ccsRewardTranslationAgeSeconds = seconds`
Description: The maximum number of seconds that CCS reward translation data remains cached.
Type: Integer
Optionality:
Allowed:
Default:
Notes:
Example: `ccsRewardTranslationAgeSeconds = 600`

ccsWalletNameTranslationAgeSeconds

Syntax: `ccsWalletNameTranslationAgeSeconds = seconds`
Description: The maximum number of seconds that CCS wallet name translation data remains cached.
Type: Integer
Optionality:
Allowed:
Default:
Notes:
Example: `ccsWalletNameTranslationAgeSeconds = 600`

ccsLanguageDetailsAgeSeconds

Syntax: `ccsLanguageDetailsAgeSeconds = seconds`
Description: The maximum number of seconds that CCS language details data remains cached.
Type: Integer
Optionality:
Allowed:

Default:
Notes:
Example: `ccsLanguageDetailsAgeSeconds = 600`

ccsBalanceTypeTranslationAgeSeconds

Syntax: `ccsBalanceTypeTranslationAgeSeconds = seconds`
Description: The maximum number of seconds that CCS balance translation data remains cached.
Type: Integer
Optionality:
Allowed:
Default:
Notes:
Example: `ccsBalanceTypeTranslationAgeSeconds = 600`

creditCardCacheAgeSeconds

Syntax: `creditCardCacheAgeSeconds = seconds`
Description: The maximum number of seconds data from the `CCS_CREDIT_CARD_DETAILS` table is cached.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 600
Notes:
Example: `creditCardCacheAgeSeconds = 600`

creditCardCacheRepIntervalSeconds

Syntax: `creditCardCacheRepIntervalSeconds = seconds`
Description: The refresh interval (in seconds) of cached credit card data.
Type: Integer
Optionality: Optional (default used if not set)
Allowed:
Default: 6
Notes:
Example: `creditCardCacheRepIntervalSeconds = 6`

defaultBalanceTypeCacheAgeSeconds

Syntax: `defaultBalanceTypeCacheAgeSeconds = seconds`
Description: The maximum number of seconds that data of the default balance type remains cached.
Type: Integer
Optionality:
Allowed:
Default: 600
Notes:
Example: `defaultBalanceTypeCacheAgeSeconds = 600`

domainsCacheAgeSeconds

Syntax: `domainsCacheAgeSeconds = seconds`

Description: The maximum number of seconds data from the `CCS_DOMAIN` table is cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `domainsCacheAgeSeconds = 600`

expenditureBalanceMidnightExpiryTZ

Syntax: `expenditureBalanceMidnightExpiryTZ = "timezone"`

Description: Daily, monthly, and yearly expenditure balances have an expiry time of midnight in the specified time zone.

Type: String

Optionality: Optional (default used if not set).

Allowed: Solaris-compliant time zone values. To view a list of accepted time zone values, see Appendix A Time Zones in the *Advanced Control Services Technical Guide*.

If this parameter is not set, expenditure balances have an expiry time of GMT midnight.

Default: GMT

Notes:

Example: `expenditureBalanceMidnightExpiryTZ = "timezone"`

ignoreBTs

Syntax: `ignoreBTs = [ type, ...]`

Description: The balance types which are not required in wallet request messages.

Type: Array

Optionality: Optional

Allowed:

Default:

Notes:

Example: `ignoreBTs = [ 201 ]`

lowCreditDapDisableCacheAgeSecondss

Syntax: `lowCreditDapDisableCacheAgeSeconds = seconds`

Description: The maximum number of seconds boolean profile tags within the `CCS_ACCT_REFERENCE.PROFILE` table are cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `lowCreditDapDisableCacheAgeSeconds = 600`

maxConcurrentChargingSessions

Syntax: `maxConcurrentChargingSessions = num`

Description: Overrides the maximum number of concurrent transactions configured for all wallets.

Type: Integer

Optionality: Optional (default used if not set).

Allowed: Values greater than or equal to 1 are valid.

Default: The value specified for the wallet is used.

Notes:

Example: `maxConcurrentChargingSessions = 50`

productTypeIdCacheAgeSeconds

Syntax: `productTypeIdCacheAgeSeconds = seconds`

Description: The maximum number of seconds product type id data from the `CCS_ACCT_TYPE` table is cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `productTypeIdCacheAgeSeconds = 600`

profileDetailsSubtagsCacheAgeSeconds

Syntax: `profileDetailsSubtagsCacheAgeSeconds = seconds`

Description: The maximum number of seconds product type data from the `CCS_ACCT_TYPE` table is cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `profileDetailsSubtagsCacheAgeSeconds = 600`

rateLimitIFSendErrors

Syntax: `rateLimitIFSendErrors = seconds`

Description: The length of time in seconds between syslog messages about not being able to send to a SLEE Interface. (BE only)

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 60

Notes:

Example: `rateLimitIFSendErrors = 600`

systemCurrencyBalanceUnitCacheAgeSeconds

Syntax: `systemCurrencyBalanceUnitCacheAgeSeconds = seconds`

Description: The maximum number of seconds that data of the system currency balance unit remains cached.

Type: Integer

Optionality:

Allowed:

Default: 600

Notes:

Example: `systemCurrencyBalanceUnitCacheAgeSeconds = 600`

systemCurrencyIdAgeSeconds

Syntax: `systemCurrencyIdAgeSeconds = seconds`

Description: The number of seconds the system currency ID will stay in the cache before being refreshed.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600

Notes:

Example: `systemCurrencyIdAgeSeconds = 300`

tariffPlanIdCacheAgeSeconds

Syntax: `tariffPlanIdCacheAgeSeconds = seconds`

Description: The maximum number of seconds tariff plan Id data from the `CCS_TARIFF_PLAN` table is cached.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 600

Notes:

Example: `tariffPlanIdCacheAgeSeconds = 600`

walletTypeCacheAgeSeconds

Syntax: `walletTypeCacheAgeSeconds = seconds`

Description: The maximum number of seconds data from the `CCS_WALLET_TYPE` table is cached.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 600

Notes:

Example: `walletTypeCacheAgeSeconds = 600`

`xmlInterfaceName`

Syntax:	`xmlInterfaceName = "`*`name`*`"`
Description:	The name of xml interface used when sending low credit DAP notifications.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"xmlIF"
Notes:	
Example:	`xmlInterfaceName = "xmlIF"`

Example

An example of the common parameter group of an eserv.config file that is used by the libccsCommon library is listed below. Comments have been removed.

```
common = {
    balanceTypeCascadeIdCacheAgeSeconds = 600
    balanceTypeCascadeCacheAgeSeconds = 600
    detailedCascadeCacheAgeSeconds = 600
    balanceTypeUnitCacheAgeSeconds = 600
    balanceTypeIdCacheAgeSeconds = 600
    defaultBalanceTypeCacheAgeSeconds = 600
    systemCurrencyBalanceUnitCacheAgeSeconds = 600

    accountCacheAgeSeconds = 600
    accountTypeBestPeriodsCacheAgeSeconds = 600
    ccsWlcAgeSeconds = 600

    ccsRewardTranslationAgeSeconds = 600
    ccsWalletNameTranslationAgeSeconds = 600
    ccsLanguageDetailsAgeSeconds = 600
    ccsBalanceTypeTranslationAgeSeconds = 600

    acsCustIdAgeSeconds = 600

    ignoreBTs = [ 201 ]
    authCB10ValidateSeed = false
    xmlInterfaceName = "xmlIF"
}
```

libccsVWARSUtils

Purpose

libccsVWARSUtils is used by beVWARS handlers and plug-ins to perform common tasks such as charges and recharges.

Example

The following example configuration shows the `ccsVWARSUtils` parameter group in the **eserv.config** file on the Voucher and Wallet Server node. Comments have been removed.

```
ccsVWARSUtils = {
    createAdditionalExpiryEdr = true
    createNonExpiringBuckets = false
    rechargePreUseAccounts = true
    rechargeTerminatedAccounts = false
```

```
    perBalanceEDRs = true
    raiseAlarmForMissingTemplates = false
    setNonExpiringBucketExpiryFromToday = false
    canReduceBucketExpiryFromToday = true
    earliestBucketExpiryPolicyFromToday = false
    useCallDurationInTimeExpenditure = true
    addBucketLevelCostsInEdr = true
}
```

Parameters

libccsVWARSUtils accepts the following parameters from the `ccsVWARSUtils` section in **eserv.config**.

Note: These parameters affect the common functions used by beVWARS handlers and plug-ins.

createAdditionalExpiryEdr

Syntax: `createAdditionalExpiryEdr = true|false`

Description: How to log EDRs when 'replace balance' is specified for any balance type defined for a voucher or voucher type recharge and is used in a WGR operation.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Two EDRs are generated: • An expiry EDR (type 3) for the balance which is being replaced, and • A recharge EDR (type 4) for the new bucket which is being created with the new value.
false	One recharge EDR is logged which records the old and new bucket values.

Default: false

Notes: The recharge EDR type may be overridden depending on calling mechanism.

For more information about WGR operations, see *VWS Technical Guide*.

Example:

```
createAdditionalExpiryEdr = true
```

createNonExpiringBuckets

Syntax: `createNonExpiringBuckets = true|false`

Description: What to do if a wallet recharge includes setting up a new bucket, but doesn't provide details of how to set the bucket's expiry date.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	If the recharge does not provide bucket expiry details, create the bucket without an expiry date.
false	If the recharge does not provide bucket expiry details, do not create the bucket.

Default: true

Notes:

Example:

```
createNonExpiringBuckets = false
```

perBalanceEDRs

Syntax: `perBalanceEDRs = true|false`

Description: Split multiple balance voucher recharge EDRs into several single balance voucher recharge EDRs.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

Default:

Notes:

Example: `perBalanceEDRs = true`

raiseAlarmForMissingTemplates

Syntax: `raiseAlarmForMissingTemplates = true|false`

Description: Specifies whether beVWARS raises an alarm when a recharge notification template is not found.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true, false

Default: false

Notes:

Example: `raiseAlarmForMissingTemplates = false`

rechargePreUseAccounts

Syntax: `rechargePreUseAccounts = true|false`

Description: Whether to allow wallets with a PreUse to be recharged.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed: true, false

Default: true

Notes: Recharging a PreUse wallet will also activate it.

Example: `rechargePreUseAccounts = false`

rechargeTerminatedAccounts

Syntax: `rechargeTerminatedAccounts = true|false`

Description: Whether or not to allow wallets with a terminated state to be recharged.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

true	Allow recharges of wallets with a terminated state to be recharged.
false	Do no allow recharges of wallets with a terminated state to be recharged.

Default: false

Notes: If this parameter is set to true, the recharge must set a wallet expiry extension value, or the wallet will expire immediately after the recharge is performed.

For more information about setting wallet expiry extension periods, see *Charging Control Services User's Guide*.

Example: `rechargeTerminatedAccounts = true`

setNonExpiringBucketExpiryFromToday

Syntax: `setNonExpiringBucketExpiryFromToday = true|false`

Description: Allows the bucket expiry date to be set for non-expiring buckets following a recharge that sets an expiry date and has an expiry extension policy of "From Today".

The expiry date is set if `setNonExpiringBucketExpiryFromToday` is set to **`true`**. If `setNonExpiringBucketExpiryFromToday` is set to **`false`**, then the expiry date is not set.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true or false

Default: false

Notes:

Example: `setNonExpiringBucketExpiryFromToday = true`

canReduceBucketExpiryFromToday

Syntax: `BucketExpiryFromToday = true | false`

Description: Whether or not libccsVWARutils overwrites a bucket's expiration date when the bucket is recharged and uses the "From Today" policy. This can be used to shorten expirations.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

- true – Overwrites the bucket's expiration date
- false – Does not overwrite the bucket's expiration date

Default: false

Notes:

Example: `canReduceBucketExpiryFromToday = true`

earliestBucketExpiryPolicyFromToday

Syntax: `earliestBucketExpiryPolicyFromToday = true | false`

Description: Whether or not libccsVWARSutils updates all buckets in the balance when applying logic from either of these parameters:

- `setNonExpiringBucketExpiryFromToday`
- `canReduceBucketExpiryFromToday`

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: true, false

Default: false

Notes:

Example: `earliestBucketExpiryPolicyFromToday = True`

useCallDurationInTimeExpenditure

Syntax: `useCallDurationInTimeExpenditure = true | false`

Description: For time expenditure balance type, whether to use total call duration or actual time balance expenses while incrementing.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:
- true – Use total call duration expenses while incrementing time expenditure balance.
- false – Use actual time balance expenses while incrementing time expenditure balance.

Default: false

Notes:

Example: `useCallDurationInTimeExpenditure = true`

addBucketLevelCostsInEdr

Syntax: `addBucketLevelCostsInEdr = true | false`

Description: Specifies whether to populate and add BUCKET_LEVEL_COSTS tag in EDR.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:
- true – Add Bucket Level Costs in EDR.
- false – Do not add Bucket Level Costs in EDR.

Default: false

Notes: The BUCKET_LEVEL_COSTS tag is added only in case of charging.

For mid-session EDR enabled scenarios, BUCKET_LEVEL_COSTS tag will be present only in the mid-session EDRs and not in the final EDR.

Example: `addBucketLevelCostsInEdr = true`

Chapter 6

Tools and Utilities

Overview

Introduction

This chapter provides a description of the operational programs or executables which are used to administer CCS. All of these processes are performed when needed.

Executables are located in the **/IN/service_packages/CCS/bin** directory.

Some executables have accompanying scripts that run the executables after performing certain cleanup functions. All scripts should be located in the same directory as the executable.

Note: Most processes can be re-started using the UNIX kill command.

Using SLP Trace log files

Processes started by the inittab and cronjobs produce logfiles that are stored in the **tmp** folder of each service directory, that is **/IN/service_packages/CCS/tmp/**.

Voucher tools

The voucher-related tools are documented in *Voucher Manager Technical Guide*.

In this chapter

This chapter contains the following topics.

ccsAccount 297
ccsBeResync 322
ccsBatchCharge 341
ccsDomainMigration 344
ccsMFileDump 349
ccsProfileBulkUpdate 352
ccsVoucherStartup.sh 353
CCS Balance Top Up Suite 353
CCS Balance Top Up MSISDN Files 355
CCS Balance Topup Rule Scripts 357
dwsublist.sh 361
Example Balance Top Up Rule Execution 363

ccsAccount

Purpose

ccsAccount enables you to generate large numbers of CCS subscribers and wallets by batch. This is a good way to create thousands of subscribers and wallets with minimal effort.

The ccsAccount tool has two modes of running:

1 Generating subscribers and wallets

2 Rolling back an unsuccessful or interrupted run

Note: This program is signal aware:

- Use SIGHUP to reload the configuration (throttling and so on)
- **Ctrl**+**C** once will stop new account generation and finish the current ones
- **Ctrl**+**C** twice will completely stop the tool

ccsAccount, when run with the `-P` (privacy) option, enables you to:

- Generate large numbers of CCS card or subscriber accounts (and corresponding wallets if none exist) randomly in a batch
- Ensure a sequential serial number is allocated and stored into the CLI
- Encrypt the print shop output file

Rollback

This tool will usually ensure that the system is not left in an inconsistent state. The rollback will remove any rows that ccsAccount cannot verify were created successfully. It will not rollback the whole batch, or even the CCS_ACCOUNT_BATCH row. Rollbacks are done by re-running the tool with the -R option (see Command line parameters).

Startup - ccsAccountStartup.sh

ccsAccountStartup.sh runs ccsAccount to generate subscriber accounts normally or to rollback account generation. **ccsAccountStartup.sh** is usually started by smsReportsDaemon when a user clicks a button on the CCS UI. However, it can be run directly from the command line by ccs_oper. On a standard installation, it runs from **/IN/service_packages/CCS/bin/**.

Startup - ccsAccountWithPrivacy.sh

ccsAccountWithPrivacy.sh can be run directly from the command line. It must be run by the user who imported the GPG key that will be used. If the key was imported using the Public Keys tab, it must be run by smf_oper. On a standard installation, it runs from **/IN/service_packages/CCS/bin/**.

Run the program in privacy mode:

```
ccsAccountWithPrivacy.sh GPG_key output_filename other_ccsAccount_parameters
```

Where:

other_ccsAccount_parameters are defined in the table in Command line parameters.

Result:

ccsAccountWithPrivacy.sh will extract the GPG key and direct the encrypted output to the print shop filename. The rest of the parameters are passed through to the ccsAccount binary as follows:

Command:

```
ccsAccount -P -m encryption_module other_ccsAccount_parameters
```

Result: The account batch output file is generated.

eserv.config parameters

The ccsAccount supports the following parameters from the `CCS` section of **eserv.config**.

Note: Some of the CCS shared parameters are also used by ccsAccount:

- *oracleUserAndPassword* (on page 52)
- accountNumberLength

accountNumberLength

Syntax:	`accountNumberLength = int`
Description:	The number of digits in card number in a subscriber account. If accountNumberLength is set to zero (0) then the account number can be any length.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	Used by ccsAccount when generating subscriber accounts.
Example:	`accountNumberLength = 14`

batchFullness

Syntax:	`batchFullness = percentage`
Description:	Sets a limit (expressed as a percentage) to control how full ccsAccount can allow the batch to become during the run.
Type:	Percentage
Optionality:	Required when ccsAccount is run in privacy mode.
Allowed:	
Default:	50
Notes:	This parameter is only applicable when the -P option is used.
Example:	`batchFullness = 90`

cardNumberIncludesServiceProviderPrefix

Syntax:	`cardNumberIncludesServiceProviderPrefix = true|false`
Description:	Determines if the service provider prefix should be included or not when writing out card numbers to the print shop file.
Type:	Boolean
Optionality:	Optional
Allowed:	true, false
Default:	true
Notes:	
Example:	`cardNumberIncludesServiceProviderPrefix = false`

ccsAccount section

Syntax:	`ccsAccount = { }`
Description:	This section contains the parameters defining ccsAccount generation config.
Type:	
Optionality:	
Allowed:	
Default:	
Notes:	
Example:	

checkAccountNumbers

Syntax: `checkAccountNumbers = true|false`

Description: Whether to check that the supplied subscriber numbers (or generated from the CLIs with the -a option) do not already exist.

Type: boolean

Optionality:

Allowed: true, false

Default: true

Notes:

Example: `checkAccountNumbers = true`

checkCLIs

Syntax: `checkCLIs = true|false`

Description: Whether to check that the supplied CLIs do not already exist.

Type: boolean

Optionality:

Allowed: true, false

Default: true

Notes:

Example: `checkCLIs = true`

doSMFChallenge

Syntax: `doSMFChallenge = true|false`

Description: Determines whether to do the SMF authentication challenge that allows the process to update the SMF database.

Type: Boolean

Optionality: Optional

Allowed: true, false

Default: true

Notes:

Example: `doSMFChallenge = true`

maximumRetries

Syntax: `maximumRetries = int`

Description: Determines the number of times the ccsAccount tool will accept, in succession, a match to an existing account number, while generating random card/account numbers.

Type: Integer

Optionality: Required when ccsAccount is run in privacy mode.

Allowed:

Default: 15

Notes:

- This parameter is only applicable when the -P option is used.
- On exhausting this value, the tool will abort the attempt, even if a unique number has not been found, thus avoiding infinite execution. This is more likely to occur as the batch approaches 100% of it's capacity.

Example: `maximumRetries = 100`

maximumSendAttempts

Syntax: `maximumSendAttempts = int`

Description: This defines the maximum number of attempts to send the wallet create request to the VWS.

Type: Integer

Optionality:

Allowed:

Default: 3

Notes:

Example: `maximumSendAttempts = 3`

noAbortOnException

Syntax: `noAbortOnException = true | false`

Description: Specifies whether ccsAccount continues to generate accounts or fails and creates a core file when an unknown SQL exception occurs during account creation.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

- true – ccsAccount continues to generate accounts after an unknown SQL exception occurs
- false – ccsAccount fails and creates a core file when an unknown SQL exception occurs

Default: false

Notes:

Example: `noAbortOnException = true`

progressUpdateInterval

Syntax: `progressUpdateInterval = seconds`

Description: The number of seconds ccsAccount should wait between writing syslog messages monitoring progress.

Type: Integer

Optionality:

Allowed:

Default: 60

Notes:

Example: `progressUpdateInterval = 60`

rollbackFilename

Syntax: `rollbackFilename = "dir"`

Description: The location of the persistent store of the wallet creation status to allow rollback of incomplete/inconsistent wallets.

Type: String

Optionality:

Allowed:

Default: "/tmp/ccsAccount-rollbackCache"

Notes:

Example: `rollbackFilename = "/tmp/ccsAccount-rollbackCache"`

sendRetryDelay

Syntax: `sendRetryDelay = seconds`
Description: The number of seconds ccsAccount should wait between retry attempts.
Type: Integer
Optionality:
Allowed:
Default: 16
Notes:
Example: `sendRetryDelay = 16`

serialNumberLength

Syntax: `serialNumberLength = int`
Description: Determines the length of the generated card serial numbers (CLIs).
Type: Integer
Optionality: Optional
Allowed: Within the range from 5 to 19
Default: 11
Notes: This includes the 2 digit service provider prefix number in its length, so a value of 11 will include `# nn000000000`.
Example: `serialNumberLength = 7`

suppressCreateWalletMes – Oberth

Syntax: `suppressCreateWalletMes = true | false`
Description: Specifies whether ccsAccount prints out the create wallet message.
Type: Boolean
Optionality: Optional (default used if not set)
Allowed:

true	Does not print out the create wallet message.
false	Prints out the create wallet message.

Default: false
Notes:
Example: `suppressCreateWalletMes = false`

wantReplicationLogging

Syntax: `wantReplicationLogging = true|false`
Description: Determines whether to tell the replication subsystem that there are changes to replicate out.
Type: Boolean
Optionality: Optional
Allowed: true, false
Default: true
Notes:
Example: `wantReplicationLogging = true`

ClientIF section

Syntax: `ClientIF {}`

Description: Section containing the parameters for the libBeClientIF.

Type: Parameter array

Optionality: Optional

Allowed: Any parameter which is supported by the libBeClientIF.

Default: Empty

Notes: For more information about the libBeClientIF, see *VWS Technical Guide*.

Example:

heartbeatPeriod

Syntax: `heartbeatPeriod = microsecs`

Description: The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.

Type: Integer

Optionality: Required

Allowed:

0	Disable heartbeat detection.
positive integer	Heartbeat period.

Default: 3000000

Notes: 1 000 000 microseconds = 1 second.

If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair.

This parameter is used by libBeClientIF.

Example: `heartbeatPeriod = 10000000`

clientName

Syntax: `clientName = "name"`

Description: The unique client name of ccsAccount.

Type: String

Optionality: Mandatory

Allowed:

Default: "ccsAccount"

Notes: The server generates clientId from a hash of the value.

If more than one client attempts to connect with the same name, then some connections will be lost.

This parameter is used by libBeClientIF. However, ccsAccount uses a different default.

Example: `clientName = "ccsAccount-uasprod01"`

connectionRetryTime

Syntax: `connectionRetryTime = seconds`

Description: The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.

Type: Integer

Optionality: Required
Allowed:
Default: 5
Notes: This parameter is used by libBeClientIF.
Example: `connectionRetryTime = 2`

maxOutstandingMessages

Syntax: `maxOutstandingMessages = num`
Description: The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.
Type: Integer
Optionality: Required
Allowed:
Default: If this parameter is not set, the maximum is unlimited.
Notes: If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls.

The messages are queued until the Voucher and Wallet Server has reduced its outstanding load.

This parameter is used by libBeClientIF.
Example: `maxOutstandingMessages = 100`

messageTimeoutSeconds

Syntax: `messageTimeoutSeconds = seconds`
Description: The time that the client process will wait for the server to respond to a request.
Type: Integer
Units: Seconds
Optionality: Required
Allowed:

1-604800	Number of seconds to wait.
0	Do not time out.

Default: 2
Notes: After the specified number of seconds, the client process will generate an exception and discard the message associated with the request.

This parameter is used by libBeClientIF.
Example: `messageTimeoutSeconds = 2`

notEndActions

Syntax:
```
notEndActions = [
    {type="str", action="[ACK|NACK]"}
    [...]
]
```
Description: The `notEndActions` parameter array is used to define the messages associated with dialogs that should not have their dialog closes, because the dialog is closed by default. This facilitates failover.
Type: Parameter array.
Optionality: Required

Allowed:

Default:

Notes: If the incoming dialog for a call closes and the last response received was of the notEndActions type, the client process sends an ABRT message. The ABRT message allows the VWS to remove the reservation. An example of this situation would be where slee_acs has stopped working.

This parameter is used by libBeClientIF.

For more information about slee_acs, see *ACS Technical Guide*.

Example:

```
notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

action

Syntax:

Description: Action to take with a message.

Type:

Optionality:

Allowed:

- "NACK"
- "ACK"

Default:

Notes:

Example:

type

The type of message.

plugins

Syntax:

```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    ...
]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Optional (as plug-ins will not be loaded if they are not configured here, this parameter must include any plug-ins which are needed to supply application functions; for more information about which plug-ins to load, see the BeClient section for the application which provides the BeClient plug-ins).

Allowed:

Default: Empty (that is, do not load any plug-ins).

Notes: The libclientBcast plug-in must be placed last in the plug-ins configuration list.
For more information about the libclientBcast plug-in, see *VWS Technical Guide*.
This parameter is used by libBeClientIF.

Example:
```
plugins = [
    {
        config="broadcastOptions"
        library="libclientBcast.so"
        function="makeBroadcastPlugin"
    }
]
```

config

Syntax: `config="name"`
Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `config="voucherRechargeOptions"`

function

Syntax: `function="str"`
Description: The function the plug-in should perform.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `function="makeVoucherRechargePlugin"`

library

Syntax: `library="lib"`
Description: The filename of the plug-in library.
Type: String
Optionality: Required (must be present to load the plug-in)
Allowed:
Default: No default
Notes:
Example: `library="libccsClientPlugins.so"`

reportPeriodSeconds

Syntax: `reportPeriodSeconds = seconds`
Description: The number of seconds separating reports of failed messages.
Type: Integer
Units: Seconds

Optionality: Required

Allowed:

Default: 10

Notes: BeClient issues a failed message report:

- For timed-out messages
- For unrequested responses
- For new calls rejected because of congestion
- For messages with invalid Voucher and Wallet Server identifiers
- If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working

VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`.

This parameter is used by libBeClientIF.

Example: `reportPeriodSeconds = 10`

throttling section

Syntax: `throttling = { }`

Description: This section contains the parameters used to minimize impact on the SMF databases and the VWSs.

Type:

Optionality:

Allowed:

Default:

Notes:

Example:

`maxAccountsPerMinute`

Syntax:

Description: This sets the unit wallet create requests per minute. Maximum rate of requests to VWSs.

Type: integer

Optionality:

Allowed:

Default: 60

Notes:

Example:

`preAllocDebug`

Syntax: `preAllocDebug = true | false`

Description: Specifies whether ccsAccount and its stored procedures create additional debug logs.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:
- true – ccsAccount and its associated stored procedures create additional debug logs
- false – ccsAccount and its associated stored procedures do not create additional debug logs

Default: false

Notes:

Example: `preAllocDebug = true`

`preVerifyAccountNumber`

Syntax: `preVerifyAccountNumber = true | false`

Description: Specifies whether ccsAccount validates that an account number does not already exist in the database before it uses the account number.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:
- true – Validates that an account number does not already exist in the database before it uses the account number. When it finds a duplicate account number during validation, ccsAccount uses the next free account number.
- false – Does not perform account number validation.

Default: false

Notes:

Example: `preVerifyAccountNumber = true`

`queueSize`

Syntax:

Description: Unit wallet requests to VWS. Maximum wallet creations in progress at once.

Type: integer

Optionality:

Allowed:

Default: 10

Notes:

Example:

BE eserv.config parameters

The following parameters are available in the `BE` section of the **eserv.config**.

`beLocationPlugin`

Syntax: `beLocationPlugin = "lib"`

Description: The plug-in library that finds the Voucher and Wallet Server details of the Voucher and Wallet Servers to connect to.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default: libGetccsBeLocation.so

Notes: This library must be in the LD_LIBRARY_PATH.

Example: `beLocationPlugin = "libGetccsBeLocation.so"`

Example ccsAccount section parameters

An example of the parameters of a SLC **eserv.config** file which configures ccsAccount. Comments have been removed.

```
CCS = {
...
accountNumberLength = 10
...
    ccsAccount = {
        maximumSendAttempts = 3
        sendRetryDelay = 16
        progressUpdateInterval = 60
        rollbackFilename = "/tmp/ccsAccount-rollbackCache"
        doSMFChallenge = true
        wantReplicationLogging = true
        serialNumberLength = 7
        maximumRetries = 100
        batchFullness = 90
        cardNumberIncludesServiceProviderPrefix = false
        checkAccountNumbers = true
        checkCLIs = true
        suppressCreateWalletMes = true
        noAbortOnException = true
        ClientIF = {
            clientName = "ccsAccount"
            plugins = []
        }
        throttling = {
            queueSize = 10
            maxAccountsPerMinute = 60
            preAllocDebug = true
            preVerifyAccountNumber = true
        }
    }
}
```

Note: ccsAccount also uses the global parameters:

- *oracleUserAndPassword* (on page 52)
- *accountNumberLength* (on page 52)
- *beLocationPlugin* (on page 135).

Seeing the configuration

You can see the configuration that the tool is running, when not in privacy mode, by setting the debug flag ccsAccount_config:

```
export DEBUG=ccsAccount_config
```

The recommended debug flags are:

```
export DEBUG=ccsAccount,ccsAccount_config
```

Normal running should not require debug flags set.

Command line parameters

ccsAccount supports the following command-line switches and parameters.

`-a`

Syntax: `-a`

Description: If specified, uses the service provider prefix + CLI provided in CLI file as the account number.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed: Set (true)

Not set (false)

Default: Not set (false)

Notes: A CLI file must be specified and the account number range is ignored. Account numbers are made up of <*Service Provider prefix* + *CLI*>.

-a and -P commands are mutually exclusive. If -a specified the -P option cannot be specified.

Example:

`-A`

Syntax: `-A`

Description: Adds the alternate number provided to accounts and activates it.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Default: Not set

Notes:

Example:

`-b`

Syntax: `-b limitType`

Description: Specifies that one of the following limit types should be used:

- Credit
- Debit
- Limited_credit
- Single_debit

Type:

Optionality:

Allowed:

Default: single_debit

Notes:

Example: `-b Debit`

`-B`

Syntax: `-B BEId`

Description: If set, specifies the Voucher and Wallet Server ID for the Voucher and Wallet Server pair the accounts will be generated to on the charging domain.

Type: Integer

Optionality: Optional

Allowed:

Default:

Notes: The charging domain can be on a VWS or a third party domain.

Example:

`-c`

Syntax: `-c cur`

Description: The three-letter currency code.

Type: String

Optionality:

Allowed: This code is checked against a list of allowable currency codes stored in the database. Must be a valid currency abbreviation, for example: NZD.

Default:

Notes:

Example:

`-C`

Syntax: `-C PINDigits`

Description: The context string for authentication type. Defines the number of digits in the PIN if the CCS1 Compatible module is used. Either `-C` or `-F` can be used, depending on the authentication module. `-C` is used where the same context string is to be used for all generated accounts. When using the CCS1 authentication module, the user may enter a specific string to indicate a specific PIN length.

Type: PIN digits

Optionality:

Allowed:

Default:

Notes:

Example: `-C 4` [indicates a four-digit PIN.]

`-d`

Syntax: `-d`

Description: Specifies direct database storage should be used.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Set (true)	Accounts generated will be written to the database and the Voucher and Wallet Server.
Not set (false)	If omitted, does not touch the database and the VWS.

Default: Not set (false)

Notes:

Example:

`-e`

Syntax: `-e endOfRange`

Description: The end account number for the account number range.

Type:

Optionality: Optional (default used if not specified.)

Allowed:

Default:

Notes:

- The end of range indicates the number after the last account number of the range.
- The number of digits in account numbers must match the *AccountLength* entry in **ccsSms.conf** and **ccsScp.conf**.
- The end of range number must be prefixed with the service provider prefix. The service provider prefix can be found by checking the CCS_RESOURCE_LIMITATION table.
- **`-e`** and **`-s`** parameters must both be present or both be absent.

Example: `-e 1000020000`

`-F`

Syntax: `-F context_file`

Description: Specifies the context file in which authentication information is stored. `-F` is used where a potentially different context string is to be used for each generated account. The system checks the file indicated.

Type: String

Optionality: Optional

Allowed:

Default:

Notes:

Example:

`-h`

Syntax: `-h`

Description: Whether to display the usage for ccsAccount.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Set (true)	A help message will be printed to stdout.
Not set (false)	The message will not be generated.

Default: Not set

Notes:

Example:

`-i`

Syntax: `-i batch_file`

Description: Specifies the location and name of the batch input file. Checks for import accounts previously exported by the ccsAccount utility.

Type:

Optionality:

Allowed:
Default:
Notes:
Example:

`-l`

Syntax: `-l CLIListFileName`
Description: The CLI list file name. The program refers to this file in order to allocate CLIs to the accounts of this batch. ccsAccount looks in the file specified and allocates CLIs to the individual accounts created.
Type:
Optionality:
Allowed:
Default:
Notes:
Example:

`-m`

Syntax: `-m pam`
Description: The authentication module name.
Type: String
Optionality: Mandatory
Allowed: DES
Default: none
Notes: Populated by the **Authentication Name** field in the New Subscriber Batch screen.
Provided by ccsVoucher_CCS3 using ccsLegacyPIN, where account number + PIN is to be used.
Example: `-m DES`

`-M`

Syntax: `-M int`
Description: The maximum number of concurrent accesses.
Type: Integer
Optionality:
Allowed:
Default: 1
Notes:
Example:

`-n`

Syntax: `-n numberOfAccounts`
Description: The number of accounts to generate.
Type: Integer
Optionality:

Allowed: The number to generate is checked to make sure it does not overflow the 32 bit unsigned integer.

Default:

Notes: If specified, the `-s` and `-e` switches are checked to make sure the account number range is large enough to cover the number of accounts to generate.

Example: `-n 5000`

`-o`

Syntax: `-o file`

Description: The file to which generated accounts are written. Can be used with the `-d` switch. If the `-d` switch is used, only accounts successfully created in the database will be inserted into the file. The rest will go to the exceptions file.

Type:

Optionality:

Allowed:

Default:

Notes:

Example:

`-O`

Syntax: `-O CLI_offset`

Description: Sets the CLI offset to use to avoid creating duplicate CLIs in parallel; for example, if the first instance of ccsAccount creates 100 accounts (`-n` is set to 100), then for the second instance of ccsAccount set `-O` to 100 to ensure there is no overlap.

Type: String

Optionality: Optional (default used if not set)

Allowed:

Default:

Notes:

Example: `-O 250`

`-p`

Syntax: `-p previous_wallet_state`

Description: Specifies the previous wallet state for the account. Must be one of:

- active
- dormant
- frozen
- preuse
- suspended
- terminated

Type: String

Optionality: Required when using the `-U` option

Allowed: active, dormant, frozen, preuse, suspended, or terminated

Default:

Notes:

Example: `-p active`

`-P`

Syntax: `-P`

Description: Privacy mode. Using privacy mode causes:

- Account numbers to be allocated randomly within the batch
- A serial number to be generated into CLI
- The exported print shop file to be encrypted

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Default: Not set (do not use privacy mode)

Notes:

- You must also set `-n`, `-s` and `-e`.
- Do not use with `-l`.
- `-a` and `-P` commands are mutually exclusive. If `-P` specified the `-a` option cannot be specified.

Example:

`-r`

Syntax: `-r`

Description: Turn off replication logging.

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Default: Log replication

Notes:

Example:

`-R`

Syntax: `-R`

Description: Performs a Voucher and Wallet Server rollback (cleanup).

Type: Boolean

Optionality: Optional (default used if not set)

Allowed:

Set (true)	Perform rollback.
Not set (false)	Do not perform rollback.

Default: Not set (false)

Notes: This is only for subscriber accounts that could be inconsistent - not the whole batch.

This option can only be specified on its own.

Example:

`-s`

Syntax: `-s` *`startOfRange`*

Description: Starting account number for the account number range.

Type:

Optionality: Optional (default used if not specified)
Allowed:
Default:
Notes:
- The start of range indicates the first number of the range.
- The number of digits in the account numbers must match the <*AccountLength*> entry in **ccsSms.conf** and **ccsScp.conf**.
- The start of range number must be prefixed with the service provider prefix. The service provider prefix can be found by checking the CCS_RESOURCE_LIMITATION table.
- **-e** and **-s** parameters must both be present or both be absent.

Example: `-e 1000010000`

`-S`

Syntax: `-S ServiceProviderName`
Description: The service provider's name.
Type:
Optionality:
Allowed:
Default:
Notes: The `-S` option takes as its argument the service provider's name, not the service provider's ID.
Example:

`-t`

Syntax: `-t type`
Description: The product type name.
Type: String
Optionality:
Allowed:
Default:
Notes: The product type is checked for validity against existing product types in the database.
Example: `-t PTS`

`-T`

Syntax: `-T trackerDomainID`
Description: Specifies the Voucher and Wallet Server ID for the Voucher and Wallet Server pair the accounts will be generated to on the tracking domain.
Type: Integer
Optionality: Optional.
Allowed:
Default:
Notes: Must be a VWS domain type.
Example:

`-u`

Syntax: `-u`

Description: When the -u option is specified, this informs ccsAccount that a VWS Voucher and Wallet Server is being used and therefore wallets can be created.

Type: Boolean

Optionality: Optional

Allowed: Set (true)

Not set (false)

Default: Not set

Notes:

Example:

`-U`

Syntax: `-U`

Description: If set, perform an update to the wallet state for a range of CLIs.

Type: Boolean

Optionality: Optional

Allowed:

Default:

Notes: Use the `-p`, `-w`, `-x`, `-y`, and `-B` options in conjuction with the `-U` option to define the CLI range, the old and new wallet states, and the Voucher and Wallet Server pair on which to perform the updates. When you use the `-U` option, the `-p` and `-w` options that specify the old and new wallet states, and the `-x` and `-y` options that define the CLI range, are required.

Example: The following example ccsAccount command sets all the accounts that have phone numbers in the range 01473640000 to 01473649999, and that are currently frozen, to active:

```
ccsAccount -U -x 01473640000 -y 01473650000 -p frozen -w active
```

`-v`

Syntax: `-v`

Description: Turns on the voice mail/alternate number generator.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

Set (true)	Use voice mail/alternate number generator.
Not set (false)	Do not generate alternate numbers.

Default: Not set (false)

Notes:

Example:

`-V`

Syntax: `-V`

Description: Generates a voice mail number and activates it.

Type: Boolean

Optionality: Optional (default used if not set).

Allowed:

Default: Not set.

Notes:

Example:

`-w`

Syntax: `-w wallet_state`

Description: Specifies the wallet state for a newly created wallet. Specifies the new wallet state for wallets updated by using the `-U` option.

Type: String

Optionality: Required when using the `-U` option to update the wallet state

Allowed: active, dormant, frozen, preuse, suspended, or terminated

Default: preuse

Notes: Use in conjuction with the `-U` option to update the wallet state.

Example: `-w dormant`

`-x`

Syntax: `-x CLI`

Description: Specifies the starting Calling Line Identifier (CLI) in a range of CLIs.

Type: String

Optionality: Required when using the `-U` option

Allowed:

Default: Not set

Notes: Use in conjunction with the `-U` option to specify the start of the update range.

Example: -x 01247640000

`-y`

Syntax: `-y CLI`

Description: Specifies the ending Calling Line Identifier (CLI) in a range of CLIs.

Type: String

Optionality: Required when using the `-U` option

Allowed:

Default: Not set

Notes: Use in conjunction with the `-U` option to specify the end of the update range.

Example: `-y 01473650000`

Example usage

```
ccsAccount -B BE Id -t prod_type [-s start -e end] -n number -b limit_type -C
PIN_digits -c currency [-d] -o file [-a] -l CLI_list_filename -m auth_module_name [-
u]
```

Exported card/account batch files

Subscriber account/calling card batch file format is controlled by the account writer plug-in used to generate the batch. Which libraries are used is defined by the authentication name specified in the New Subscriber Batch screen.

Header fields are in the format "*Key_field_name*=*value*". Key field names always start with an alphabetic character. This makes it easy to distinguish them from voucher records (which always start with a number).

The following header fields are used in the voucher batch file header, (although downstream processors should detect any "*Key_field_name*=*value*" lines).

Header field	Description
AccountBatchID=*int*	The ID of the subscriber account batch.
ServiceProviderID=*int*	The ID number of the service provider the subscriber batch belongs to. When ccsAccount is started by the screens the value of this field is populated by the id of the service provider which is selected in the **Service Provider** field of the Subscriber Management screen when the **New** button is clicked.
AccountTypeID=*int*	The product type the subscriber batch has. When ccsAccount is started by the screens the value of this field is populated by the **Product Type** field on the New Subscriber Batch screen.
maxConcurrent=*int*	The maximum number of concurrent connections wallets generated with this subscriber batch can have. When ccsAccount is started by the screens the value of this field is populated by the **Maximum Concurrent Accesses** field on the New Subscriber Batch screen.
BatchSize=*int*	The number of subscriber accounts in this batch. When ccsAccount is started by the screens the value of this field is populated by the **Batch Size** field on the New Subscriber Batch screen.
RangeStart=*int*	Beginning of the range of subscriber account numbers. When ccsAccount is started by the screens the value of this field is populated by the **Card Number Start Range** field on the New Subscriber Batch screen.
RangeEnd=*int*	End of the range of subscriber account numbers. When ccsAccount is started by the screens the value of this field is populated by the **Card Number End Range** field on the New Subscriber Batch screen.
AuthenticationModuleID=*int*	The ID of the authentication module used for: • Encryption and/or random generation of PINs for this batch • (optionally) sends the output file for encryption by gpg. When ccsAccount is started by the screens the value of this field is populated by the **PAM Name** field on the New Subscriber Batch screen.
BillingEngineID=*int*	The ID number of the Voucher and Wallet Servers .
CurrencyID=*int*	The ID of the currency the wallets generated with this subscriber batch will use. When ccsAccount is started by the screens the value of this field is populated by the **Wallet Currency** field on the New Subscriber Batch screen.

Header field	Description
LimitType=*str*	The type of limit the wallets generated with this subscriber batch will use.
BalanceType=*int*	The balance type ID of the balance type this wallet will have any initial value stored in.

A line consisting of a single equal sign (=) terminates the header lines. All subsequent lines are voucher detail records.

ccsAccount example

This table gives an example of how to use ccsAccount to generate a batch of subscriber accounts.

Stage	Description
1	User telnets to the SMS on which the CCS application is installed and logs in as ccs_oper.
2	User navigates to the directory in which ccsAccount is located. In a standard installation, this will be **/IN/service_packages/CCS/bin**.
3	User starts the subscriber account batch generation with the following command: `$ ccsAccount -t PTS -s 141000 -e 141100 -n 10 -b debit -C 4 -c EUR -d -o /tmp/AcctGenDemo.txt -l Sample_CLI_file`

4	ccsAccount generates 10 subscribers and wallets of "PTS" product type.
	Subscriber numbers start at 141000 and end at 141100. PIN length limit is 4. The subscribers and wallets are stored in the database, the output file is **/tmp/AcctGenDemo.txt**, the EUR currency will be used and the wallets will have debit balances.
	Note: AccountLength must be configured to be “6” for this to succeed.
	The `-l cli_filename` parameter specifies the CLI file to which the ccsAccount program will refer. The CLI file you specify contains lines which are either:
	• Comments (start with #)
	• Blank
	• Single CLI (may not start with 0, must be a number)
	• Range of CLIs (neither beginning nor end may start with 0, beginning and end separated by '-')
	Example:
	`# Sample CLI file#`
	`95551212`
	`95550000-95550050`
	`955500595559999`
	`95053333`
	The ccsAccount program goes through the lines one at a time. If a range is given, it goes through the range in order. Both the beginning and end of a range are available for use as CLIs for a subscriber.

Tip: Information may appear on the screen in DEBUG builds of ccsAccount to show the progress of the subscriber/wallet generation. However, if this information is not displayed during the generation process, a summary of it may be viewed by using the output command. A subset of this information will appear when using a non-debug build.

Example card/account output file

Here is an example ccsAccount command and the resulting account batch output file:

Command

```
ccsAccount -P -t "World" -m "DES" -s 8815000000 -e 8820990000 -n 10 -b debit
-C 7 -c USD -d  2>&1
```

Card/account output file

This text shows an example export subscriber account/calling card output file.

```
# Account Batch Output File
# Generated Wed Dec 31 01:24:29 2008
#
AccountBatchID=59
ServiceProviderID=1
AccountTypeID=7
maxConcurrent=1
BatchSize=10
RangeStart=8815000000
RangeEnd=8819990000
AuthenticationModuleID=4
BillingEngineID=2
CurrencyID=2
LimitType=DEBT
BalanceType=1
=
Dec 31 01:24:29.861203 ccsAccount(15179) NOTICE: Beginning account generation.
16309877,3415992,7,G8.H3zCjoKzbY,8800127
19052821,0363266,7,G8fRbQy015unk,8800128
18627603,5447142,7,G82efn9Gh2gSY,8800129
16635167,9003194,7,G8nkF67MOzS9g,8800130
19498256,8441931,7,G8tfZtbQvbOIg,8800131
18758105,8744644,7,G8CSYLULMZtww,8800132
17349265,3517347,7,G8GH/BM14HHzs,8800133
16223817,0064708,7,G8MbgIe4gPO.U,8800134
16089674,7771756,7,G8lXd7ySSzsVw,8800135
16405822,1207166,7,G8JugOSguxjqg,8800136
Dec 31 01:24:35.514685 ccsAccount(15179) NOTICE: Progress 10/10 (100.0%) Complete
Dec 31 01:24:35.515578 ccsAccount(15179) NOTICE: Account generation complete.
```

Failure

If ccsAccount fails, the accounts may not have been created correctly. Use the rollback function to tidy up the miscreated accounts. Rerun the tool.

Output

ccsAccount writes a log of all created subscriber accounts to **/IN/service_packages/CCS/account/export/**.

ccsAccount inserts data into the following tables in the SMF:

- CCS_ACCT
- CCS_ACCT_REFERENCE
- CCS_ACCT_ACCT_REFERENCES
- CCS_ACCT_HIST_INFO

Other ccsAccount commands

The following file allows the user to view the actual output of the file, whether or not that information was displayed during the Account generation.

```
$ cat /tmp/AccountDemo.txt
```

This allows the user to take input from a file and insert it into the database.

Note: When a limited credit account is created, the minimum credit balance is set by default to = 0. The minimum credit balance must then be reset manually to the required amount, using the SMS Java administration screens.

Example

```
# Account Batch Output File
# Generated Mon Aug 28 01:15:52 2000
#
AccountBatchID=0
ServiceProviderID=22
AccountTypeID=35
BatchSize=10
RangeStart=141500
RangeEnd=141520
AuthenticationModuleID=1
BillingEngineID=1
CurrencyID=1
BalanceType=2
=

141500,0801,4,G8bVdVSGtI.9.
141501,4742,4,G8WI1B6IHdSQI
141502,6891,4,G8ACBmfc.cYGg
141503,9394,4,G8OVlG4MDKtmQ
141504,4904,4,G8iiqCNLGD./k
141505,9709,4,G8JoxdWtgYmkk
141506,0158,4,G8uGhZ4LG5qfE
141507,2641,4,G8o6Lc/./i/uw
141508,1468,4,G8/wyTezMlx9U
141509,9023,4,G8JMbJcWiem1E
$ ccsAccount-d -i filename -c EUR
```

ccsBeResync

Purpose

The ccsBeResync is a tool that will resolve and/or report differences between Voucher and Wallet Servers in a logical pair for wallet and voucher tables that may be caused by a software or hardware fault. It does not replace the beSync, which is used to keep the pair synchronized during normal operation.

The tool has two primary operational modes:

1. Resynchronizing wallets
2. Resynchronizing vouchers

Without a command line option specifying vouchers it will default to resynchronizing wallets. In addition, the tool can be run to one of the following:

- Use one specified Voucher and Wallet Server in a pair as the model data source (that is, when the data on the other Voucher and Wallet Server has been corrupted or is out of date)
- Use configurable business rules to make updates on both Voucher and Wallet Servers

A range of wallet or voucher IDs can be specified, so as to limit the range of voucher/wallets it processes as described in the configuration section.

The ccsBeResync tool can be configured with both:

- Command line arguments
- Entries in the `ccsBeResync` section of the **eserv.config** file

Data transferred

This table lists the E2BE database columns for which the ccsBeResync tool will resynchronize E2BE data.

Table	Column
BE_WALLET	.NEVER_EXPIRES .EXPIRY .MAX_CONCURRENT .STATE .NEVER_ACTIVATED .ACTIVATION_DATE
BE_BALANCE	.LIMIT_TYPE .MINIMUM_CREDIT
BE_BUCKET	.EXPIRY .NEVER_EXPIRES .VALUE .LAST_USE .NEVER_USED .START_DATE .REFERENCE
BE_VOUCHER	.REDEEMED .REDEEMED_DATE .REDEEMING_WALLET_ID

Startup

It is recommended that ccsBeResync is run from the SMS but it can be run from anywhere so long as it is able to connect to the SMF and E2BE databases.

Command Line parameters

Running ccsBeResync from the command line with the -h flag will print out a list of the command line parameters.

Usage:

```
ccsBeResync -u username/password [-b id] [-m primary|secondary] [-r] [-v] [-s id] [-e id] [-o filename] [-h] [-?]
```

The available parameters are:

`-u`

Syntax: `-u username/password`

Description: VWS database username and password.

Type: String

Optionality:

Allowed:

Default:

Notes:

Example: `-u e2be_admin/manager`

`-b`

Syntax: `-b id`

Description: The ID of the Voucher and Wallet Server pair to synchronize.

Type:

Optionality:

Allowed:

Default:

Notes: This should be the same as the entry from CCS_BE_LOCATION.BE_ACCT_ENGINE_ID for logical Voucher and Wallet Server pair.

Example:

`-e`

Syntax: `-e id`

Description: End of range, using one of the following:

- CCS_ACCT.BE_ACCT_ID if synchronizing wallets (default)
- CCS_VOUCHER_REFERENCE.ID when synchronizing vouchers (-v). Defaults to 0.

`-h/-?`

Display syntax help.

`-m`

Syntax: `-m primary|secondary`

Description: Master [primary|secondary]. The Voucher and Wallet Server which will be the master data source.

Type:

Optionality:

Allowed:

- primary
- secondary

Default:

Notes: If specified, will only make changes to the slave Voucher and Wallet Server regardless of the nature of the differences that ccsBeResync identifies. Records that only exist on the slave will not be recreated on the master and all updates resulting from differences will be resolved in favor of the master.

Example:

`-o`

Syntax: `o filename`

Description: When specified, ccsBeResync will redirect individual difference and overall statistics output to the nominated file.

Type:

Optionality:

Allowed:

Default: stdout

Notes:

Example:

`-r`

Syntax:

Description: Report mode.

Type:

Optionality:

Allowed:

Default:

Notes: Wallet/voucher information requests are sent and differences reported either as standard output or in the output file (**-o** parameter).

No updates will be made to either Voucher and Wallet Server.

Example:

`-s`

Syntax: `-s id`

Description: Start of range, using one of the following:

- CCS_ACCT.BE_ACCT_ID if synchronizing wallets (default)
- CCS_VOUCHER_REFERENCE.ID when synchronizing vouchers (-v). Defaults to 0.

`-v`

Syntax:

Description: Vouchers. Will cause ccsBeResync to perform synchronization updates/reporting to be performed on vouchers instead of wallets.

Type:

Optionality:

Allowed:

Default:

Notes: If this is not set it will default to performing the synchronization on wallets only. The tool will not attempt doing both.

Example:

Configuration - eserv.config

ccsBeResync supports these parameters from the `ccsBeResync` section of the **eserv.config** file.

```
ccsBeResync = {
    beID = id
    syncSequenceDifference = int
    maxQueueSize = int
    pollTime = seconds
    recheckDelay = seconds
    maxInfoRechecks = int
    maxUpdatesPerRequest = int
    beRequestTimeout = seconds
    notificationInterval = seconds

    bucketValueHighest = true|false
    bucketExpiryLatest = true|false
```

```
    balanceMinCreditHighest = true|false
    balanceLimitTypeHighest = true|false
    walletMaxConHighest = true|false
    walletExpiryLatest = true|false
    walletActivationLatest = true|false
    bucketReferencePrimary = true|false
    skipLastUseFieldCheckAtWallet = true|false

    beLocationPlugin = "lib"
    oracleUserPass = "usr/pwd"
    clientName = "name"

    heartbeatPeriod = microsecs
    messageTimeoutSeconds = seconds
    maxOutstandingMessages = int
    reportPeriodSeconds = seconds
    connectionRetryTime = seconds

    plugins = [
        {
            config="confStr",
            library="lib",
            function="str"
        }
        [...]
    ]

    confStr = {
        plugin configuration
    }

    notEndActions = [
        {type="str", action="[ACK |NACK]"}
        [...]
    ]
}
```

Parameters

The ccsBeResync supports the following general parameters from the CCS section of **eserv.config**.

beID

Syntax:	`beID = id`
Description:	ID of the Voucher and Wallet Server pair to resynchronize.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1
Notes:	This value can be found in the database table CCS_DOMAIN.DOMAIN_ID. Only one pair can be re-synchronized at a time.
Example:	`beID = 3`

beRequestTimeout

Syntax:	`beRequestTimeout = seconds`
Description:	The number of seconds to wait before timing out a message and giving up on that particular wallet/voucher.
Type:	integer

Optionality:	Optional
Allowed:	
Default:	60
Notes:	This will produce syslog messages.
Example:	`beRequestTimeout = 60`

maxInfoRechecks

Syntax:	`maxInfoRechecks = int`
Description:	If when the wallet/voucher is rechecked for differences, it has changed; it will wait `recheckDelay` and try again.
Type:	integer
Optionality:	Optional
Allowed:	
Default:	5
Notes:	If it is a heavy use wallet, ccsBeResync will give up after maxInfoRechecks.
Example:	`maxInfoRechecks = 5`

maxQueueSize

Syntax:	`maxQueueSize = int`
Description:	Maximum number of wallets/ vouchers being checked simultaneously.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	10
Notes:	Increasing this setting will have an impact on the VWSs, but in turn will make resynchronization faster. The count: • Includes all wallets/vouchers with outstanding WI/VI and updates • Excludes the wallets/vouchers sleeping/waiting to do another WI/VI in order to confirm the differences before correction
Example:	`maxQueueSize = 10`

maxUpdatesPerRequest

Syntax:	`maxUpdatesPerRequest = int`
Description:	Maximum size of updates for VWSs. If an update is larger than this maximum, the message will be split into more than one part.
Type:	integer
Optionality:	Optional
Allowed:	
Default:	10
Notes:	Large updates sent to the Voucher and Wallet Server may cause problems with the size of a SLEE event. Instead, send maxUpdatesPerRequest each time, and send more than one request if we have more than that many updates to send. ccsBeResync will log alarms if timeouts are occurring because updates are too large.

Example: `maxUpdatesPerRequest = 10`

notificationInterval

Syntax: `notificationInterval = seconds`
Description: How often, in seconds, to print out the progress of the ccsBeResync tool.
Type: integer
Optionality:
Allowed:
Default: 300
Notes:
Example: `notificationInterval = 300`

pollTime

Syntax: `pollTime = seconds`
Description: Maximum number of seconds to poll VWS connections before attempting to start another request or check sleeping requests.
Type: integer
Optionality: Optional
Allowed:
Default: 1
Notes:
Example: `pollTime = 1`

recheckDelay

Syntax: `recheckDelay = seconds`
Description: Number of seconds wait before rechecking an inconsistent wallet/voucher record.
Type: integer
Optionality: Optional
Allowed:
Default: 120
Notes: Setting this too low will cause transactions to be applied twice; once by this tool, and again by the transaction from the other VWS.

This should be at least the time it takes for a transaction to make it from one VWS to the other.

Inconsistent records are/can be caused when a record has been updated on one VWS, but not synced with the other by beSync yet.
Example: `recheckDelay = 120`

skipLastUseFieldCheckAtWallet

Syntax: `skipLastUseFieldCheckAtWallet = true|false`
Description: If this parameter is set to true, ccsBeResync tool synchronizes the wallets with active reservation where commits are not happening during `recheckDelay` interval for that reservation.
Type: Boolean
Optionality: Optional
Allowed:

Default:	false
Notes:	Only those wallets are synchronized whose LUSE field is changing between two WI_Ack responses received for the requests sent `recheckDelay` seconds apart to a particular VWS node.
Example:	`skipLastUseFieldCheckAtWallet = true`

`syncSequenceDifference`

Syntax:	`syncSequenceDifference = int`
Description:	The maximum allowable difference between sequence numbers on the Voucher and Wallet Servers. If this amount is exceeded, the tool will abort the resynchronization. This prevents ccsBeResync from applying transactions twice (once itself, and once by the beSync).
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	Negative integer — Ignore any different between sequence numbers. This is useful if (for example) you are recreating the entire database after hardware failure. Positive integer — Maximum allowable difference before aborting the resync
Default:	10
Example:	`syncSequenceDifference = 10`

eserv.config business rules parameters

The ccsBeResync tool recreates any rows deemed missing in BE_WALLET, BE_BALANCE, BE_BUCKET on either VWS. Therefore, the following parameters are set to determine the resolution of differences between rows that exist on both Voucher and Wallet Servers.

Note: If the following parameters are not defined, the defaults will be applied to the row, that is, even if the parameter is not set, the row will be updated with the default behavior.

`balanceLimitTypeHighest`

Syntax:	`balanceLimitTypeHighest = true|false`
Description:	Make both balances have the same value by taking the highest (true)/ lowest (false) value. Defined (lowest->highest) order is: • SingleUse • Debit • LimitedCredit • Credit
Type:	boolean
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	false
Notes:	Alters BE_BALANCE.LIMIT_ TYPE.
Example:	`balanceLimitTypeHighest = false`

balanceMinCreditHighest

Syntax:	`balanceMinCreditHighest = true|false`
Description:	Make both balances have the same minimum credit limit by taking the highest (true)/ lowest (false) value.
Type:	boolean
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	false
Notes:	Alters BE_BALANCE.MINIMUM_ CREDIT.
Example:	`balanceMinCreditHighest = false`

bucketExpiryLatest

Syntax:	`bucketExpiryLatest = true|false`
Description:	Makes both buckets have the same expiry by taking the earliest (true)/ latest (false) expiry.
Type:	boolean
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	true
Notes:	The latest possible expiry is 'never expires'. Alters BE_BUCKET.EXPIRY and BE_BUCKET.NEVER_EXPIRES.
Example:	`bucketExpiryLatest = true`

bucketReferencePrimary

Syntax:	`bucketReferencePrimary = true|false`
Description:	Which VWS Voucher and Wallet Server to use as the master data when resynchronizing buckets (BE_BUCKET) which have a reference and start date.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Use the values from the primary VWS Voucher and Wallet Server to set the periodic charges. false — Use the values from the secondary VWS Voucher and Wallet Server to set the periodic charges.
Default:	true
Notes:	Applies to periodic charge buckets (that is, periodic charges).
Example:	`bucketReferencePrimary = false`

bucketValueHighest

Syntax:	`bucketValueHighest = true|false`
Description:	Makes both buckets have the same value by taking the highest (true)/ lowest (false) value.
Type:	boolean
Optionality:	Optional, default will be used if not specified
Allowed:	true, false
Default:	true
Notes:	Alters BE_BUCKET.VALUE

Example:	`bucketValueHighest = true`

`walletActivationLatest`

Syntax:	`walletActivationLatest = true|false`
Description:	Make both wallet have the same activation date by taking the earliest (true)/ latest (false) expiry.
Type:	
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	true
Notes:	The latest possible expiry is 'never expires'. Alters BE_WALLET. ACTIVATION_ DATE.
Example:	`walletActivationLatest = true`

`walletExpiryLatest`

Syntax:	`walletExpiryLatest = true|false`
Description:	Make both wallet have the same expiry by taking the earliest (true)/ latest (false) expiry.
Type:	boolean
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	true
Notes:	The latest possible expiry is 'never expires'. Alters BE_WALLET.EXPIRY and BE_WALLET.NEVER_EXPIRES.
Example:	`walletExpiryLatest = true`

`walletMaxConHighest`

Syntax:	`walletMaxConHighest = true|false`
Description:	Make both wallets have the same maximum concurrent users by taking the highest (true)/ lowest (false) value.
Type:	boolean
Optionality:	Optional, default will be used if not specified.
Allowed:	true, false
Default:	true
Notes:	Alters BE_WALLET.MAX_ CONCURRENT.
Example:	`walletMaxConHighest = true`

libBeClientIF parameters

The ccsBeResync tool may use the libBeClientIF to connect to the Voucher and Wallet Server. The standard configuration is available in the parameters described below.

`clientName`

Syntax:	`clientName = "name"`
Description:	The unique client name of the process.
Type:	String

Optionality:	Required
Allowed:	Must be unique.
Default:	The host name of the local machine.
Notes:	The server generates clientId from a hash of *str*. If more than one client attempts to connect with the same name, then some connections will be lost. This parameter is used by libBeClientIF.
Example:	`clientName = "scpClient"`

connectionRetryTime

Syntax:	`connectionRetryTime = seconds`
Description:	The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	5
Notes:	This parameter is used by libBeClientIF.
Example:	`connectionRetryTime = 2`

heartbeatPeriod

Syntax:	`heartbeatPeriod = microsecs`
Description:	The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.
Type:	Integer
Optionality:	Required
Allowed:	0 Disable heartbeat detection. positive integer Heartbeat period.
Default:	3000000
Notes:	1 000 000 microseconds = 1 second. If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair. This parameter is used by libBeClientIF.
Example:	`heartbeatPeriod = 10000000`

maxOutstandingMessages

Syntax:	`maxOutstandingMessages = num`
Description:	The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	If this parameter is not set, the maximum is unlimited.

Notes: If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls.

The messages are queued until the Voucher and Wallet Server has reduced its outstanding load.

This parameter is used by libBeClientIF.

Example:
```
maxOutstandingMessages = 100
```

messageTimeoutSeconds

Syntax:
```
messageTimeoutSeconds = seconds
```

Description: The time that the client process will wait for the server to respond to a request.

Type: Integer

Units: Seconds

Optionality: Required

Allowed:

1-604800	Number of seconds to wait.
0	Do not time out.

Default: 2

Notes: After the specified number of seconds, the client process will generate an exception and discard the message associated with the request.

This parameter is used by libBeClientIF.

Example:
```
messageTimeoutSeconds = 2
```

notEndActions

Syntax:
```
notEndActions = [
    {type="str", action="[ACK|NACK]"}
    [...]
]
```

Description: The `notEndActions` parameter array is used to define the messages associated with dialogs that should not have their dialog closes, because the dialog is closed by default. This facilitates failover.

Type: Parameter array.

Optionality: Required

Allowed:

Default:

Notes: If the incoming dialog for a call closes and the last response received was of the notEndActions type, the client process sends an ABRT message. The ABRT message allows the VWS to remove the reservation. An example of this situation would be where slee_acs has stopped working.

This parameter is used by libBeClientIF.

For more information about slee_acs, see *ACS Technical Guide*.

Example:

```
notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

action

Syntax:

Description: Action to take with a message.

Type:

Optionality:

Allowed:

- "NACK"
- "ACK"

Default:

Notes:

Example:

type

The type of message.

plugins

Syntax:

```
plugins = [
    {
        config=""
        library="lib"
        function="str"
    }
    ...
]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Optional (as plug-ins will not be loaded if they are not configured here, this parameter must include any plug-ins which are needed to supply application functions; for more information about which plug-ins to load, see the `BeClient` section for the application which provides the BeClient plug-ins).

Allowed:

Default: Empty (that is, do not load any plug-ins).

Notes: The libclientBcast plug-in must be placed last in the plug-ins configuration list.

For more information about the libclientBcast plug-in, see libclientBcast.

This parameter is used by libBeClientIF.

Example:

```
plugins = [
    {
        config="broadcastOptions"
        library="libclientBcast.so"
        function="makeBroadcastPlugin"
    }
]
```

config

Syntax: `config="name"`

Description: The name of the configuration section for this plug-in. This corresponds to a configuration section within the `plugins` section in the **eserv.config** file.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `config="voucherRechargeOptions"`

function

Syntax: `function="str"`

Description: The function the plug-in should perform.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `function="makeVoucherRechargePlugin"`

library

Syntax: `library="lib"`

Description: The filename of the plug-in library.

Type: String

Optionality: Required (must be present to load the plug-in)

Allowed:

Default: No default

Notes:

Example: `library="libccsClientPlugins.so"`

reportPeriodSeconds

Syntax: `reportPeriodSeconds = seconds`

Description: The number of seconds separating reports of failed messages.

Type: Integer

Units: Seconds

Optionality: Required

Allowed:

Default: 10

Notes: BeClient issues a failed message report:

- For timed-out messages
- For unrequested responses
- For new calls rejected because of congestion
- For messages with invalid Voucher and Wallet Server identifiers
- If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working

VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`.

This parameter is used by libBeClientIF.

Example: `reportPeriodSeconds = 10`

Example configuration

An example of the `ccsBeResync` parameter group of a SLC **eserv.config** file is listed below. Comments have been removed.

```
ccsBeResync = {
    beID = 1
    syncSequenceDifference = 10
    maxQueueSize = 10
    pollTime = 1
    recheckDelay = 120
    maxInfoRechecks = 5
    maxUpdatesPerRequest = 10
    beRequestTimeout = 60
    notificationInterval = 300

    bucketValueHighest = true
    bucketExpiryLatest = true
    balanceMinCreditHighest = false
    balanceLimitTypeHighest = false
    walletMaxConHighest = true
    walletExpiryLatest = true
    walletActivationLatest = true
    bucketReferencePrimary = true
    skipLastUseFieldCheckAtWallet = false
}
```

Failure

Re-synchronization between the data in the source E2BE database and the data in the destination E2BE database will fail. Any discrepancies between the databases may remain. Rerun the tool.

Output

Rows that do not exist on one VWS will be created on the other (only on the slave if performing a master/slave resynchronization).

The ccsBeResync tool will first establish a connection to the Voucher and Wallet Servers in the specified VWS pair. It will then send wallet/voucher information requests to both Voucher and Wallet Servers.

- If the responses do not match it will wait for a configurable number of seconds and send requests again. This is to determine whether the data it is querying is currently in use and waiting for normal synchronization processing to complete.

- If the responses from the first and second queries do not match (that is, the differences between the wallets have changed since the first information request), it keeps trying until it receives a matching response from subsequent requests.
- If no master has been specified it then creates updates according to the business rules set in the **eserv.config** file and sends them to the appropriate Voucher and Wallet Servers.
- If a master has been specified ('primary'/'secondary'), it will only create updates that will force the slave Voucher and Wallet Server data to become a duplicate of the master.

Note: The updates are a special message that will not be subject to the normal synchronization process, that is, after being sent to one or the other Voucher and Wallet Server they will not be duplicated across the pair after they have been applied.

Resynchronizing in Normal Operation

The tool is installed in **/IN/service_packages/CCS/bin**. To run the ccsBeResync tool you must ensure that the ccsBeResync section is present in the **eserv.config** file that you are using. The tool should be run by ccs_oper. Output of differences found between the Voucher and Wallet Servers when using business rule definitions will be sent to the system log in the form of a NOTICE entry as well as to the tool's standard output. When performing master/slave resynchronizations the differences will only be sent to standard output or the output file.

Examples of normal operation:

```
$/IN/service_packages/CCS/bin/ccsBeResync
```

With no command line options selected the ccsBeResync tool will:

- Use the Voucher and Wallet Server pair specified in the **eserv.config** file
- Use business rules to resynchronize records
- Process wallets only
- Output to stdout (no report file will be created)
- Check and update wallet-related database columns specified below for all records in those tables (no start or end range defined).

Table	Column
BE_WALLET	.NEVER_EXPIRES .EXPIRY
BE_BALANCE	.LIMIT_TYPE .MINIMUM_CREDIT
BE_BUCKET	.MAX_CONCURRENT .STATE .NEVER_EXPIRES .EXPIRY .NEVER_ACTIVATED .ACTIVATION_DATE .VALUE

```
$/IN/service_packages/CCS/bin/ccsBeResync -r -m secondary -o
/tmp/Wallet_Resync_Report.txt
```

With the above command line options the ccsBeResync tool will:

- Use the Voucher and Wallet Server pair specified in the **eserv.config** file
- Create a report only. No updates to the databases on either VWS will be performed
- Process wallets only

- Create and write output to **/tmp/Resync_Report.txt** (note: this file will be overwritten by re-running the ccsBeResync tool unless another filename is specified)
- Check and report on wallet-related database column differences for ALL records in those tables (no start or end range defined)

```
$/IN/service_packages/CCS/bin/ccsBeResync -v -s 1000 -e 5000 -o
/tmp/Voucher_Resync.txt
```

With the above command line options the ccsBeResync tool will:

- Use the Voucher and Wallet Server pair specified in the **eserv.config** file
- Process vouchers only
- Check and update the BE_VOUCHER.REDEEMED database column according to business rules in the **eserv.config** file for records with IDs between BE_VOUCHER.ID 1000 and BE_VOUCHER.ID 5000
- Create and write output to **/tmp/Voucher_Resync.txt**

Normal error conditions

The ccsBeResync tool will exit on certain error conditions before it has been able to process all records. These include:

- ccsBeResync process killed during processing
- Configuration file parsing errors
- Command line parsing errors
- Unable to connect to one or both Voucher and Wallet Servers:
  - Database unavailable
  - Voucher and Wallet Server not running or disabled
  - Connection to database or Voucher and Wallet Server broken
  - Voucher and Wallet Servers too far out of sync (configurable with override)

When the ccsBeResync has been interrupted during processing the statistics output will report how far through the selected list of records the tool had reached, for example:

Statistics:

- Completed IDs = 3579
- In sync vouchers = 3579
- Last ID processed = 280525
- Total IDs = 100020
- Voucher info acks = 3579
- Voucher info requests sent to primary VWS = 3589
- Voucher info requests sent to secondary VWS = 3589

Note: Statistics not listed were equal to zero.

Resynchronization Reports

The standard report will contain configuration information used by the ccsBeResync tool, any differences between the specified E2BE databases that were found and a statistics summary for all actions taken by the tool during processing.

Example:

```
ccsBeResync starting on Fri Oct 3 11:03:55 2003

ccsBeResync Configuration
-------------------------
 beID                   : 1
 masterBE               : not defined
 syncSequenceDifference : -1
```

```
 startRange             : 0
 endRange               : 0
 smfUserPass            : /
 Primary BE IP          : 192.168.0.191
 Primary BE Port        : 1700
 Secondary BE IP        : 192.168.0.190
 Secondary BE Port      : 1700
 BE Oracle SID          : E2BE
 BE Oracle logon        : e2be_admin/e2be_admin
 Max Queue size         : 10
 Poll Time              : 2
 Recheck Delay          : 10
 Max Info Rechecks      : 5
 BE Request Timeout     : 60
 Notification Interval  : 3
 Output filename        : syncWallet.out
 No master defined, using business rules

Config map for first BE beClientIF = {
    billingEngines = [
        {
            id = 1
            primary = {
                ip = "192.168.0.191"
                port = 1700
            }

        }

    ]

    clientName = "ccsBeResync"
    plugins = []
}

Config map for second BE beClientIF = {
    billingEngines = [
        {
            id = 1
            primary = {
                ip = "192.168.0.190"
                port = 1700
            }

        }

    ]

    clientName = "ccsBeResync"
    plugins = []
}

 Process wallets
 Report and fix inconsistencies
 -----------------------------------
 Business rules
  Highest Bucket Value             : true
  Highest Bucket Expiry Date       : true
  Highest Min Credit Value         : true
  Highest Limit Type               : true
  Highest Wallet Max Concurrent    : true
  Highest Wallet Expiry Date       : true
  Highest Wallet Activation Date   : true
```

```
 ----------------------------------

Updating primary BE wallet 144 maxCon:1->1 state:PREU->ACTV neverExpires:1->1
expiryDate:0->0 neverActivated:1->1 activationDate=0->0
Wallet 282 Updating secondary bucket 90080 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 284 Updating secondary bucket 90084 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 286 Updating secondary bucket 90088 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 288 Updating secondary bucket 90092 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 290 Updating secondary bucket 90096 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Updating primary BE wallet 281 maxCon:1->3 state:ACTV->ACTV neverExpires:1->1
expiryDate:0->0 neverActivated:0->0 activationDate=1064964017->1064964017
Wallet 281 Updating secondary bucket 90078 Value 102530100->105291100 Expiry
neverExpires->neverExpires
Wallet 283 Updating secondary bucket 90082 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 285 Updating secondary bucket 90086 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 287 Updating secondary bucket 90090 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 292 Updating secondary bucket 90100 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 294 Updating secondary bucket 90102 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 296 Updating secondary bucket 90106 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 298 Updating secondary bucket 90112 Value 102442000->105181000 Expiry
neverExpires->neverExpires
Wallet 291 Updating secondary bucket 90098 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 293 Updating secondary bucket 90104 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 295 Updating secondary bucket 90108 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 297 Updating secondary bucket 90110 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 299 Updating secondary bucket 90116 Value 102420000->105181000 Expiry
neverExpires->neverExpires
Wallet 300 Updating secondary bucket 90114 Value 102442000->105192000 Expiry
neverExpires->neverExpires

Statistics:
    BE_WALLET rows updated = 2
    Completed IDs = 5027
    In sync wallets = 5000
    Last ID processed = 5280
    Secondary BE_BUCKET rows updated = 19
    Total IDs = 5027
    Update Acks = 21
    Update responses received = 21
    Updates sent = 21
    Updates sent to primary = 2
    Updates sent to secondary = 19
    Wallet Info Acks = 5027
    Wallet Info Requests sent to primary BE = 5060
    Wallet Info Requests sent to secondary BE = 5060
    Wallet diffs without updates = 7
    Wallet that changed, and required checking again = 6
    Wallets checked second time = 33
```

```
ccsBeResync stopped at Fri Oct  3 11:05:19 2003
```

ccsBatchCharge

Purpose

The ccsBatchCharge tool applies batches of updates to subscriber wallets.

ccsBatchCharge permits the activation, execution and deactivation of rules that are used to allocate additional items to a specified balance type for selected subscribers.

Example

ccsBatchCharge supports the following command line parameters:

```
ccsBatchCharge [-i file] [-o file] [-c str] [-h] [-?]
```

Parameters

ccsBatchCharge accepts the following command line parameters.

`-b`

Syntax:	`-b bucket`
Description:	Default bucket (if not specified in input).
Type:	integer
Optionality:	
Allowed:	
Default:	-1
Notes:	
Example:	

`-c`

Syntax:	`-c str`
Description:	The section of the **eserv.config** file to get configuration for bePlugin.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	BE
Notes:	
Example:	`-c`

`-d`

Syntax:	`-d debitstrategy`
Description:	Debit strategy rule selection.
Type:	integer
Optionality:	
Allowed:	1 = SINGLE_NO_NEG 2 = SINGLE_NEG 3 = MULTIPLE

Default: 1
Notes:
Example:

`-e`

Syntax: `-e CDRextrainfovalue`
Description: Extra information to put into EDR in cdrExtraInfoTag.
Type: string
Optionality:
Allowed:
Default: CCSBC
Notes:
Example:

`-h`

Displays the help file.

`-i`

Syntax: `-i file`
Description: File to read batch information from.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: stdin
Notes:
Example: `-i ChargeBatch.txt`

`-m`

Syntax: `-m maxpending`
Description: Maximum number of requests pending at any time.
Type: integer
Optionality:
Allowed:
Default: 10
Notes:
Example:

`-o`

Syntax: `-o file`
Description: The file to write error output to.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: stdout
Notes:
Example: `-o ChargeBatch.log`

`-p`

Syntax: `-p seconds`
Description: Default poll time for beClient.
Type: integer
Optionality:
Allowed:
Default: 1
Notes: in seconds.
Example:

`-r`

Syntax: `-r num`
Description: Number of times to poll a request before timing it out.
Type: integer
Optionality:
Allowed:
Default: 30
Notes:
Example:

`-t`

Syntax: `-t balancetype`
Description: Default balance type (if not specified in input).
Type: string
Optionality:
Allowed:
Default: 'General Cash'
Notes:
Example:

`-w`

Syntax: `-w wallettype`
Description: Wallet type.
Type:3 string
Optionality:
Allowed:
Default: 'Personal'
Notes:
Example:

`bePlugin`

Syntax:
Description: Override the default config section used to get information about bePlugin.

Type:	string
Optionality:	
Allowed:	
Default:	beLocationPlugin
Notes:	
Example:	

`cdrExtraInfoTag`

Syntax:	
Description:	Name of the tag added to the EDR which holds extra information configured in cdrExtraInfoValue.
Type:	string
Optionality:	
Allowed:	
Default:	CCSBC
Notes:	
Example:	

`-?`

Displays the help file.

ccsDomainMigration

Purpose

ccsDomainMigration takes details from the SMS screens and migrates wallets between VWS Voucher and Wallet Servers. For more information about migrating wallets, see *VWS Technical Guide*.

Startup

Start `ccsDomainMigration` from the Service Management System (SMS) by selecting the **Services->Prepaid Charging->Subscriber Management** screen and clicking **Restart** on the **UBE Account Balancing** tab. When you push the **Restart** button, SMS passes parameters to `ccsDomainMigration` which is started by the `ccsDomainMigrationStartup.sh` script.

For more information about the **UBE Account Balancing** tab, see *Charging Control Services User's Guide*.

You can also invoke the `ccsDomainMigrationStartup.sh` script in test mode to test connectivity to the VWS servers that are involved in a migration. In this case, the script does not actually perform a migration.

Note: Invoke `ccsDomainMigrationStartup.sh` as a command *only* if wish to run a connectivity test. Use the **Restart** button in the **UBE Account Balancing** tab to perform an actual migration.

To run `ccsDomainMigration` in test mode, invoke it from the command line with the `-t` parameter. In test mode, `ccsDomainMigration` reports to the log file whether it successfully connects to the VWS servers.

You can also specify the `-p pollTime` option in conjunction with the `-t` option to set the poll time to use in the connection test. The `polltime` parameter value overrides the value of the `polltime` parameter in the **eserv.config** file. After connecting to a VWS, `ccsDomainMigration` sends four polls before sending the first wallet migration request. The `-p polltime` option specifies the number of seconds that `ccsDomainMigration` waits after sending each poll.

The `-p` option allows you to test different poll time values to determine which ones are best suited for connecting to the VWS. You then can use those values for the `polltime` parameter in the **eserv.config** file.

Configuration

ccsDomainMigration supports parameters from the `ccsDomainMigration` parameter group in the **eserv.config** file on a SMS. It contains parameters arranged in the structure shown below.

```
ccsDomainMigration = {
    ClientIF = {

        heartbeatPeriod = microsecs
        messageTimeoutSeconds = secs
        maximumSendAttempts = int
        pollTime = secs
        progressTimeout = secs
        sendRetryDelay = secs
    }
    lockFile = "dir"
    commitInterval = int
    commitVolume = int
    throttle = int
}
```

Note: ccsDomainMigration also uses the global parameters:

- *beLocationPlugin* (on page 135)
- *oracleUserAndPassword* (on page 52)

Parameters

ccsDomainMigration supports the following parameters in the `ccsDomainMigration` section of **eserv.config**.

`ClientIF section`

Syntax:	`ClientIF {}`
Description:	Section containing the parameters for the libBeClientIF.
Type:	Parameter group
Optionality:	Required
Allowed:	Any parameter which is supported by the libBeClientIF.
Default:	Empty
Notes:	For more information about the libBeClientIF, see *VWS Technical Guide*.
Example:	

`heartbeatPeriod`

Syntax:	`heartbeatPeriod = microsecs`	
Description:	The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.	
Type:	Integer	
Optionality:	Required	
Allowed:	0	Disable heartbeat detection.
	positive integer	Heartbeat period.

Default:	3000000
Notes:	1 000 000 microseconds = 1 second. If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair. This parameter is used by libBeClientIF.
Example:	`heartbeatPeriod = 10000000`

`messageTimeoutSeconds`

Syntax:	`messageTimeoutSeconds = seconds`
Description:	The time that the client process will wait for the server to respond to a request.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	1-604800 Number of seconds to wait. 0 Do not time out.
Default:	2
Notes:	After the specified number of seconds, the client process will generate an exception and discard the message associated with the request. This parameter is used by libBeClientIF.
Example:	`messageTimeoutSeconds = 2`

`maximumSendAttempts`

Syntax:	`maximumSendAttempts = num`
Description:	The number of times that a particular wallet request will be sent to a VWS. A request is resent if either an unrecoverable error occurs or the request times out. The first request sent is counted as attempt number one.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	3
Notes:	If you specify a value that is lower than the default, `ccsDomainMigration` uses the default value instead.
Example:	`maximumSendAttempts = 5`

`pollTime`

Syntax:	`pollTime = seconds`
Description:	The number of seconds between the four Voucher and Wallet Server polls ccsDomainMigration makes after it has made a connection to the Voucher and Wallet Server before sending the first wallet migration request.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1
Notes:	The time spent polling enables the beServer and ccsDomainMigration to establish a confirmed connection. If errors appear in the syslog indicating a connection has been established and request sending is failing, this value should be increased.

Example: `pollTime = 2`

progressTimeout

Syntax: `progressTimeout = seconds`

Description: The number of seconds between checks to ensure that a migration is making progress, that is, that wallet requests are being processed.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 120

Notes: If you specify a value that is lower than the default, `ccsDomainMigration` uses the default value instead.

Example: `progressTimeout = 180`

sendRetryDelay

Syntax: `sendRetryDelay = seconds`

Description: The number of seconds between attempts to send a wallet request. The limit on the number of attempts is specified by the `maximumSendAttempts` parameter.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 16

Notes: If you specify a value that is lower than the default, ccsDomainMigration uses the default value instead.

Example: `sendRetryDelay = 20`

commitInterval

Syntax: `commitInterval = seconds`

Description: The maximum number of seconds between wallet update commits to the SMF.

Type: Integer

Optionality: Optional (default used if not set).

Allowed: Positive integers

Default: 15

Notes: Wallet update commits may also be triggered by the number of commits exceeding *commitVolume* (on page 347).

Example: `commitInterval = 15`

commitVolume

Syntax: `commitVolume = int`

Description: The number of records to commit in one batch.

Type: Integer

Optionality: Optional (default used if not set).

Allowed:

Default: 200

Notes:	Wallet update commits may also be triggered by the number of seconds between commits exceeding *commitInterval* (on page 347).
Example:	`commitVolume = 200`

`lockFile`

Syntax:	`lockFile = "`*`path`*`"`
Description:	The directory path and filename of the lockfile.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"/IN/service_packages/CCS/tmp/ccsDomainMigration.lock"
Notes:	On a clustered SMS this must be on the global file system. On a clustered SMS this should be set to the same value on all nodes.
Example:	`lockFile = "/IN/service_packages/CCS/tmp/ccsDomainMigration.lock"`

`throttle`

Syntax:	`throttle = `*`int`
Description:	The maximum number of wallet migration requests to send to the VWS each second.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	positive integer maximum requests 0 no limit (disable throttling)
Default:	0
Notes:	For each migration, the lowest value of this setting and the setting in the Throttle field for the migration is used.
Example:	`throttle = 2`

`walletLockMilliSec`

Syntax:	`walletLockMilliSec = `*`milliseconds`
Description:	The number of milliseconds during which the migrating subscriber's wallets will be locked on the source VWS while being migrated to the destination VWS.
Type:	Integer 32 bit integer value (signed)
Optionality:	Optional
Default:	30000
Example:	`walletLockMilliSec = 30000`

Example

An example of the `ccsDomainMigration` parameter group of a Voucher and Wallet Server **eserv.config** file is listed below. Comments have been removed.

```
ccsDomainMigration = {
    ClientIF = {
        heartbeatPeriod = 10000000
        messageTimeoutSeconds = 2
        maximumSendAttempts = 5
```

```
        pollTime = 1
        progressTimeout = 180
        sendRetryDelay = 20
    }
    lockFile = "/IN/service_packages/CCS/tmp/ccsDM.lock"
    commitInterval = 10
    commitVolume = 100
    throttle = 2
}
```

Failure

If ccsDomainMigration fails on startup, the **UBE Account Balancing** tab will report an error and no changes will be made.

If ccsDomainMigration fails or is stopped while processing a migration, ccsDomainMigration will exit and attempt to commit any pending successful transactions to the SMF database. However, it is likely that some wallets will have been migrated on the Voucher and Wallet Server, but the confirmation has not been returned to the ccsDomainMigration process so the SMF database will not reflect those changes. ccsDomainMigration should not be stopped manually. Instead, the migration should be stopped using the **Cancel** or **Pause** buttons on the **UBE Account Balancing** tab. For more information about the **UBE Account Balancing** tab, see *Charging Control Services User's Guide*.

Output

ccsDomainMigration updates wallet location and migration details in the following tables in the SMF database.

Note: You can use the **UBE Account Balancing** tab in the Subscriber Management screen to export the migration report to a flat file. For more information, see *Charging Control Services User's Guide*.

The ccsDomainMigration writes error messages to the system messages file, and also writes additional output to **/IN/service_packages/CCS/tmp/ccsDomainMigration.log**.

ccsMFileDump

Purpose

ccsMFileDump writes data from a specified binary MFile into formatted text or html.

Startup

ccsMFileDump is started from the command line.

Configuration

ccsMFileDump supports the following command-line switches and parameters.

```
ccsMFileDump [-h|-H prefix] [-c CLI -d DN [-t timestamp] [-p str]] file
```

`file`

Syntax:	*file*
Description:	The name of the CCS MFile to validate and dump. For named event catalogue MFiles the filename must begin with 'P'.
Type:	String
Optionality:	Mandatory

Allowed:
Default:
Notes:
Example: `001160095644`

`-c`

Syntax: `-c` *CLI*
Description: Dump a portion of the MFile only for the specified CLI and DN.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default: Dump information for all CLI.
Notes: If `-c` is specified, `-d` should also be specified.
Example: `-c 03`

`-d`

Syntax: `-d` *DN*
Description: Dump a portion of the MFile only for the specified CLI and DN.
Type: Integer
Optionality: Optional (default used if not set).
Allowed:
Default: Dump information for all destination numbers.
Notes: If `-d` is specified, `-c` should also be specified.
Example: `-d 06`

`-h`

Syntax: `-h`
Description: Output the dump in an HTML file with links.
Type: Boolean
Optionality: Optional (default used if not set).
Allowed:
Default: Dump to raw text.
Notes:
Example:

`-H`

Syntax: `-H` *prefix*
Description: Dump output to multiple HTML files.
Type: String
Optionality: Optional (default used if not set).
Allowed:
Default: Dump to raw files.
Notes: Format of file will start with:
prefix-**1234.html**
The numbers correspond to the offsets into the MFile.

Example:	`-H MFileDump`

`-p`

Syntax:	`-p product|named_event_catalogue`
Description:	Dump a portion of the MFile for the specified product or named event catalogue. The internal ID for the product type/named event catalogue must be specified. *product* is also equal to the Account Type ID in the account type section in the MFile.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	Dump data for all product types.
Notes:	For rating Mfiles, if `-p` is specified, `-c` and `-d` should also be specified. For named event catalogue Mfiles, `-p` is the only optional parameter.
Example:	`-p 4`

`-t`

Syntax:	`-t timestamp`
Description:	Dump a portion of the MFile for the given timestamp.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	The timestamp can be specified in any of the following formats: • *YYYYMMDDHHMMSS*, *YYYYMMDDHHMM* or *YYYYMMDD* • *YYMMDDHHMMSS*, *YYMMDDHHMM* or *YYMMDD*
Default:	Dump all Dates and Times for the specified CLI and DN.
Notes:	If -t is specified, `-c` and `-d` should also be specified.
Example:	`-t 20061225132500`

Rating example

These lines show examples of the command line configuration for a rating MFile (where the MFile filename is **\001160095644**):

```
ccsMFileDump 001160095644"
ccsMFileDump -h 001160095644"
ccsMFileDump -H out 001160095644"
ccsMFileDump -c 03 -d 06 001160095644"
ccsMFileDump -c 03 -d 06 -t 20061225132500 001160095644"
ccsMFileDump -c 03 -d 06 -t 20061225132500 -p 4 001160095644"
```

Named event catalogue example

These lines show examples of the command line configuration for a named event catalogue MFile (where the MFile filename is **\P001160095644**):

```
ccsMFileDump P001160095644
ccsMFileDump -h P001160095644
ccsMFileDump -H out 001160095644
ccsMFileDump -p 55 001160095644
```

Output

The ccsMFileDump writes error messages to the system messages file, and writes the content of the MFile to stdout.

ccsProfileBulkUpdate

Purpose

The ccsProfileBulkUpdate tool applies bulk updates to CCS subscriber profile field tags. It is used to update tags for integer and date profile fields. Multiple tags may be processed at the same time.

When a profile field tag is updated for a subscriber, the old profile tag is removed from the subscriber's profile and the new tag is added. The value previously associated with the old tag is assigned to the new tag.

Note: If the new tag is already present in the subscriber's profile then no changes are made to the tag.

Startup

Follow these steps to run the Profile Tags Bulk Update tool.

Step	Action
1	Login to the SMS as ccs_oper.
2	Navigate to the directory in which ccsProfileBulkUpdate is located. In a standard installation, this will be **/IN/service_packages/CCS/bin**.
3	Run the program: `ccsProfileBulkUpdate` *`parameters`* **Where:** The available *`parameters`* are defined in the table in *Command line parameters* (on page 352). **Note:** The **profileTags.cfg** configuration file is located in **/IN/service_packages/CCS/etc .**

Example

```
ccsProfileBulkUpdate [-f "filename"] [-?]
```

Command line parameters

`-f`

Syntax:	`-f "`*`filename`*`"`
Description:	The name of the input file containing the profile tag updates.
Type:	String
Optionality:	
Allowed:	
Default:	**profileTags.cfg**
Notes:	The **profileTags.cfg** configuration file is located in **/IN/service_packages/CCS/etc**.
Example:	`-f "profileTags.cfg"`

`-u <user>/<password>`

Syntax:	`-u "user/password"`
Description:	The Oracle username and password.
Type:	String
Optionality:	
Allowed:	
Default:	"/"
Notes:	
Example:	-u "/"

`-?`

Displays the help file.

Profile tags input file

The profile tags input file (**profileTags.cfg**) lists the profile tags to be updated. Each line in the file contains two decimal numbers separated by a space. These are the number for the tag to be changed followed by the number for its new tag.

Example profileTags.cfg

This is an example **profileTags.cfg** file.

```
3146497 3146498
3146511 3146512
1310724 1310725
```

ccsVoucherStartup.sh

License

The **ccsVoucherStartup.sh** script is only available if you have purchased the Voucher Management license. For more information about this tool, see *Voucher Manager Technical Guide*.

CCS Balance Top Up Suite

Introduction

CCS Balance Top Up Suite uses rules to increment balances on a regular basis. The additional balances are used in the same way as normal funds when the customer makes calls. Updates are applied to a specified balance type of the nominated subscriber wallets by the ccsBatchCharge tool.

Each promotion has associated with it:

- A rule that defines the balance to update, the frequency, the first execution date
- An MSISDN file that defines which subscriber wallets are impacted by the rule

Possible uses of ccsBatchCharge

You can use the CCSBT when you want to give a list of subscribers one of the following:

- Five notifications every week for six months and the notifications would expire one week after being added if not used
- A one-off increase of 30 units of currency that would expire one month after being added if not used

Rule definition

A rule is used to decide:

- What balance type to add to
- When to add to the balance
- How often to add to the balance (for a recurrent rule)
- How long the addition will last

Column definition

The columns allowed in the definition of a balance topup rule are detailed in the following table.

Column	Definition
Name	Name of the rule.
Item count	Number of items (or amount) to add to the balance for every execution of the rule. Valid values are * and positive and negative integers. Where the value is *, the value will be taken from an MSISDN list file. When the rule relates to non-cash balances, the value to be added is absolute (for example, for a value of 10, the number of items to be added will be 10). When the rule relates to cash balances, the value to be added is expressed as 'littles' (for example, adding a value of 15023 will result in a currency amount of 150.23).
First execution date	Date from which the rule begins execution. Valid values are: • * • Any valid date in the format *DD/MM/YYYY*. Where the value is *, the value is defaulted to the current date. In this case, the execution mode must be set to IMM. A rule with this value will fire at the next rules execution cycle.
Cycle period	The frequency that the rule fires. The cycle period value has the format nu, where: • n is a positive integer • u is the time unit. (This can be either d (days) or m (months).) **Examples:** 13d (13 days), 1d (1 day), 1m (1 month). A value of zero is allowable when iteration count is equal to 1.
Expiry period	The length of time the newly added bucket lasts. The bucket expiry date will be set, and the bucket will be removed when this date is reached. The expiry period value has the format nu, where: • n is a positive integer • u is the unit. (This can be either d (days) or m (months).) **Examples:** 13d (13 days), 1d (1 day), 1m (1 month). The expiry date on the added bucket will be date the rule is executed plus the expiry period.

Column	Definition
Iteration count	The number of times the rule is executed. This value must be 1 or more.
Execution mode	Determines whether the rule is to be executed immediately, or is to be scheduled for nightly processing. Valid values are: • IMM for immediate execution • REC for recurrent execution
Wallet type	This is the type of wallet in which the balance is incremented. This value must match a value from CCS_WALLET_TYPE.NAME **Examples:** 'Primary', 'Secondary'
Balance type	This is the type of balance that is incremented. The balance type must be a free SMS balance type and must match a value from CCS_BALANCE_TYPE.NAME.

The four functions

There are four types of basic function related to balance top-up rules.

- Activate rule
- Deactivate rule
- Deactivation cleanup
- Execute rule

Each of these functions is implemented as a separate Unix shell script on the SMS platform. The shell scripts invoke PL/SQL scripts and the PI (PIbatch) to implement the rule. For details on these rules see *CCS Balance Topup Rules scripts* (on page 357).

CCS Balance Top Up MSISDN Files

Introduction

MSISDN files contain lists of MSISDN numbers or ranges, and are used in the activation and deactivation of Balance Top Up rules.

The MSISDN file structure for activation and deactivation are the same, except that activation files must have a header record.

Note: The header record is not required for deactivation files.

Record types

There are two record types for the MSISDN file:

- Header record
- MSISDN detail record

Header record

This record type can occur only once in the file. It must be the first record in the file and it must have the following format:

```
M;text
```

This table describes MSISDN header records.

Field	Description
M	This is the amount or value to be added to the specified balance of each MSISDN account for every execution of the rule.
text	The first character of this item indicates the format of the content. The format can be: **Fixed (f)**: where the amount added by the rule is fixed and determined by the field M. **Variable (v)**: where the amount added by the rule is variable and determined by the input file content for each MSISDN in the command line. If the variable amount is blank for the MSISDN, then the amount is determined by the field M.

MSISDN detail record

This record type can occur multiple times. It must have the following format:

```
L M
```

This table describes MSISDN detail records.

Field	Description
L	This is either a single MSISDN or a range of MSISDNs. A range of MSISDNs is represented by two MSISDNS separated by a hyphen.
M	This is the amount or value to be added to the specified balance of the MSISDN account for every execution of the rule. This can only be used if the format specified in the header record is variable (v). It is only relevant where the file is used for rule activation.

Example MSISDN files for activations

Hereare example MSISDN files for activations:

```
7;fPROM56
32496556500
32496556509
32496550000-32496550020

0;vPROM90
32496556500 5
32496556509 10
32496550000-32496550020 4
32496560000-32496560020 8
```

Example MSISDN files for deactivations

Here are example MSISDN files for deactivations:

```
32496556500
32496556509
32496550000-32496550020

32496556500
32496556509
32496550000-32496550010
32496560000-32496560010
```

CCS Balance Topup Rule Scripts

Purpose

The CCS balance topup rule scripts are used to apply balance topup rules to balances. There are four scripts that are installed into **/IN/service_packages/CCS/bin** and are used in the following ways:

Use	Script
Activate rule	**ccsbt_activate_rule.sh**
Deactivate rule	**ccsbt_deactivate_rule.sh**
Deactivation Cleanup	**ccsbt_deactivate_cleanup.sh**
Execute rule	**ccsbt_execute.sh**

Activate rule

Before a rule can be executed, the operator must activate it by initiating the Activate rule script. The activation checks that the rule definition and subscriber list (MSISDN file) are valid. If they are valid, the details are stored. All rules being activated must have an associated MSISDN file.

Activation is required for rules of both immediate execution and recurrent execution modes.

A recurrent (REC) rule can only be activated once. The activation process automatically schedules the execution of the rule.

An immediate (IMM) rule where the first execution date is '*' can be activated multiple times. Reactivation of an Immediate rule replaces all MSISDNs that are currently associated with the rule with those contained in the associated MSISDN file.

ccsbt_activate_rule.sh

Use the **ccsbt_activate_rule.sh** shell script to activate CCS balance topup rules. Before running the script, log on to the SMS as ccs_oper and change to the shell script's directory.

The script must be run using the following parameters:

Usage:

`ccsbt_activate_rule.sh` *`RuleParameters MSISDNFile`* [*`user/password`*]

The available parameters are:

`Rule Parameters`

Syntax:	
Description:	The parameter definition of the rule to be activated.
Type:	
Optionality:	
Allowed:	For more information, see *Column definition* (on page 354).
Default:	
Notes:	Rule parameters must be specified in the order that they appear in the rule definition table. They must be comma separated and enclosed within single quote marks.
Example:	

MSISDN file

Syntax:
Description: The name of the file that holds the subscriber list.
Type:
Optionality:
Allowed:
Default:
Notes:
Example:

user/password

Syntax: *oracleuser/password*
Description: The Oracle user name and password to be used when running the script.
Type:
Optionality: Optional (default used if not specified).
Allowed:
Default: "/"
Notes:
Example:

Example ccsbt_activate_rule.sh

```
ccsbt_activate_rule.sh 'PROMO1,1,20/03/2005,1d,1m,2,REC,Personal,Free SMS'
PROMO1MSISDNfile
```

Note: It is recommended that you review the log file generated by the rule activation process.

Deactivate rule

Deactivating a rule lets you remove MSISDNs that are associated with it.

Depending on the MSISDNs specified, the rule can be totally deactivated, or can become non-active for certain MSISDNs to which it previously applied.

To deactivate a rule, the operator initiates the deactivate rule script. This checks that the rule name and a subscriber list (MSISDN file) are valid. If they are valid, the specified MSISDN associations are removed from the rule details.

The deactivation of a rule can only take place where the rule has already been activated.

If the deactivation of a rule removes all associations between a rule and any subscribers, then the rule record is removed from the SMF database.

ccsbt_deactivate_rule.sh

Use the **ccsbt_deactivate_rule.sh** shell script to deactivate a rule.

Before running the script, log on to the SMS as ccs_oper and change to the shell script's directory.

The script must be run using the following parameters:

Usage:

```
ccsbt_deactivate_rule.sh RuleName MSISDNFile [user/password]
```

The available parameters are:

`Rule Name`

The unique name of the rule to be deactivated.

Default: -

`MSISDN file`

The name of the file that holds the subscriber list to be deactivated.

Default: -

Note: You specify the name only. The system assumes that the file is in the **../input** directory.

`user/password`

Syntax: *oracleuser/password*

Description: The Oracle user name and password to be used when running the script.

Type:

Optionality: Optional (default used if not specified).

Allowed:

Default: "/"

Notes:

Example:

Example ccsbt_deactivate_rule.sh

This text shows an example of the **ccsbt_deactivate_rule.sh** being used.

```
ccsbt_deactivate_rule.sh PROMO1 PROMO1deactivate
```

Note: It is recommended that you review the log file generated by the rule. activation process.

Execute rules

The execute rule function adds an amount or value to subscriber balances for active rules. The rules are assessed for execution based on the first execution date, the cycle period and the iteration count.

The execute rule function is initiated automatically by two scheduled tasks:

1. The first scheduled task processes recurrent rules. This is initiated once per day, at a configurable time (the default time is 02.00 hrs). The task invokes a process that applies the rules of execution mode ‘REC’ to the relevant balances.
2. The second scheduled task processes Immediate rules. This is initiated once per hour, configurable after installation (the default is between 10:00 hrs and 17:00 hrs). The task invokes a process that applies the rules of execution mode ‘IMM’ to the relevant balances.

Immediate rules

Immediate rules only execute once after each activation.

An Immediate rule, where the first execution date is ‘*’, can be re-activated multiple times with different subscriber lists (MSISDN files). Reactivating this sort of rule replaces all MSISDNs that are currently associated with the rule, with the rules contained in the new MSISDN file. Reactivation of this sort of rule more than once per day is not supported. The execution applies to one activation that day only.

Output files

Each rule execution scheduled task generates several output files. These are:

- Log file - a log file is created for each rule execution scheduled task. You are recommended to review this file
- Daily result file - a separate daily result file is created for each execution of each rule
- Daily error file - a separate daily error file is created for each execution of each rule

Execution log file

A log file is created for each execution of all the current rules. This usually happens hourly.

The file name has the following format:

execute_rule_*rundate_runtime_machine_node*.**log**

where:

- *rundate* is the run date of the execution in *DDMMYY* format
- *runtime* is the run time of the execution in *HHMM* format
- *machine_node* is the machine node where the execution took place

The file is written to by the CCSBT software and by the ccsBatchCharge program. All activation output and ccsBatchCharge normal and error output is written to this file. After the CCSBT header information, there will be some ccsBatchCharge header information, and then one line for each MSISDN being recharged.

A successful recharge consists of the line number, the word "SUCCESS" and then the input that was used for ccsBatchCharge.

Example: `1,SUCCESS,1231,-50,Free SMS,-2,,AD`

Daily error file

A separate daily error file is created for each execution of each rule.

The file name has the following format:

ccsbt_error_*machine_node_rundate_rulename*.**err**

where:

- *machine_node* is the machine node where the execution took place
- *rundate* is the run date of the execution in *DDMMYY* format
- *rulename* is the name of the rule to which the error file pertains

Deactivation Cleanup

Deactivation cleanup provides the ability to:

1. Remove the association with a rule where the subscriber has been terminated.
2. Remove rules where the final execution date (last active date) has passed. The final execution date is the last date on which a rule executes.

The deactivation cleanup function is initiated automatically by a scheduled task.

The deactivation of a rule for a subscriber can only take place where the rule has already been activated for the subscriber.

The deactivation determines the subscribers that have been terminated and disassociates all rules from the subscriber.

REC execution mode

Rules which have the recurrent (REC) execution mode are executed in chronological order based on their first execution date. They are executed when they meet the following conditions:

Execution	Test Conditions
First execution	When the first execution date is equal to the current date.
Subsequent executions	When the: • Iteration count is greater than 1 • Current date is an iteration anniversary The rule executes 'iteration count' times, with the interval between executions determined by the cycle period. This means the rule executes if the current date is one of the dates calculated as: (first execution date + (cycle period * (1.. Iteration count - 1)))
Final execution	When the current date is equal to the following anniversary date: (first execution date + (cycle period * (Iteration count)))

IMM execution mode

Rules which have the immediate (IMM) execution mode are executed when they meet the following conditions:

Execution	Test Conditions
First execution	When the first execution date is: • Equal to '*', or the current date • The rule has not been executed since activation
Final execution	When the current date is equal to the following anniversary date: (first execution date + (cycle period * (iteration count))) **Note:** This only applies where the first execution date is a valid date.

dwsublist.sh

Purpose

The script **dwsublist.sh** is a report generating tool used to collate the account balances of each subscriber. To generate report data from your Oracle database the script uses the configurable parameters in the **dwsublist.cfg** file to connect and extract subscriber balance information. See *Parameters* (on page 362) for more information about configuring the tool.

The script is located in the **/IN/service_packages/SMS/input/Ccs_Service/Summary/dwsublis**t.Errors from the tool are written to the **dwsublist.log**.

Process

Here is a description of process that **dwsublist.sh** performs.

Stage	Description
1	Create links to each primary E2BE database.
2	Extract and merge SMF and E2BE data for each VWS.

Stage	Description
3	Process data extracted into temporary global table.
4	Fix date inconsistencies in extracted data.
5	Update CCS_ACCT_HIST_INFO.LAST_CHANGE_STATE_REASON to simulate state changes if account is dormant for a configurable period.
6	Make CCARD and PCARD temporary files (**.tmp**).
7	Process data.
8	Change CCARD and PCARD filenames from **.tmp** to real name for the system to pick up.

Reports

The dwsublist is used to collate data which can be presented in the following reports:

- Account Balance in text format
- Account Balance in CSV format

Refer to *Charging Control Services User's Guide* for details.

Parameters

The **dwsublist.sh** supports the following parameters from the **dwsublist.cfg** configuration file.

`pcardOutputDir`

Syntax:	`balancesOutputDir='`*`path`*`'`
Description:	The path for the balance output.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	'/IN/service_packages/SMS/output/Ccs_Service/Summary'
Notes:	
Example:	`balancesOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary'`

`ccardOutputDir`

Syntax:	`ccardOutputDir='`*`path`*`'`
Description:	The path to output the CCARD file.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	'/IN/service_packages/SMS/output/Ccs_Service/Summary/ccard'
Notes:	
Example:	`ccardOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary/ccard'`

`pcardOutputDir`

Syntax:	`pccardOutputDir='`*`path`*`'`
Description:	The path to output the PCARD file.
Type:	String
Optionality:	Optional (default used if not set).

Allowed:	
Default:	'/IN/service_packages/SMS/output/Ccs_Service/Summary/pcard'
Notes:	
Example:	`ccardOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary/pcard'`

Example configuration

Here is an example **dwsublist.cfg** file.

```
ccardOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary/ccard'
pcardOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary/pcard'
balancesOutputDir='/IN/service_packages/SMS/output/Ccs_Service/Summary'
```

Example Balance Top Up Rule Execution

Introduction

The following topics provide some examples of valid and invalid rule executions.

The comma separated rule consists of these components:

- Rule name
- No of SMS (*n*)
- First execution date (*a*)
- Cycle period
- Expiry period (*e*)
- Iteration count
- Execution mode (IMM or REC)
- Wallet type (*w*)
- Balance type (*b*)

In the examples the acceptable values for the following variables are:

- *p* must be an integer greater than 0
- *t* must be an integer greater than 0
- *n* must be an integer greater than 0
- *a* must be a date in the format *DD/MM/YYYY*, and it must be equal to or greater than the date of activation

Note: The parameters for each example rule are specified in the order that they appear in the rule definition table. For details, see *Column definition* (on page 354).

Valid IMM rule examples

The following table provides examples of valid immediate (IMM) rule executions.

Rule	Description
PROM_01,*n*,*,0,*e*M,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value (*n*) for each valid MSISDN, and is valid for (*e*) months.

Rule	Description
PROM_02,*,*,0,*e*M,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value determined from the MSISDN file for each valid MSISDN, and is valid for (*e*) months.
PROM_03,*n*,*a*,0,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value(*n*) for each valid MSISDN, and is valid for (*e*) months.
PROM_04,*,*a*,0,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value determined from the MSISDN file for each valid MSISDN, and is valid for (*e*) months.
PROM_05,*n*,*,0,*e*M,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value(*n*) for each valid MSISDN, and is valid for (*e*) days.
PROM_06,*,*,0,*e*M,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value determined from the MSISDN file for each valid MSISDN, and is valid for (*e*) days.
PROM_07,*n*,*a*,0,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value(*n*) for each valid MSISDN, and is valid for (*e*) days.
PROM_08,*,*a*,0,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value determined from the MSISDN file for each valid MSISDN, and is valid for (*e*) days.
PROM_09,*n*,*,*t*D,*e*M,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value(*n*) for each valid MSISDN, and is valid for (*e*) months. Cycle period is ignored.
PROM_10,*,*,*t*M,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value determined from MSISDN file for each valid MSISDN, and is valid for (*e*) months. Cycle period is ignored.
PROM_11,*n*,*a*,*t*D,*e*M,1,IMM,*w*,*b*	Execute once after date (*a*). (*b*) bucket added to (*w*) wallet with value (*n*) for each valid MSISDN, and is valid for (*e*) months. Cycle period is ignored.
PROM_12,*,*a*,*t*M,*e*M,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value determined from MSISDN file for each valid MSISDN, and is valid for (*e*) months. Cycle period is ignored.
PROM_13,*n*,*,*t*D,*e*D,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value(*n*) for each valid MSISDN, and is valid for (*e*) days. Cycle period is ignored.

Rule	Description
PROM_14,*,*,*t*M,*e*D,1,IMM,*w*,*b*	Execute once after activation. (*b*) bucket added to (*w*) wallet with value determined from MSISDN file for each valid MSISDN, and is valid for (*e*) days. Cycle period is ignored.
PROM_15,*n*,*a*,*t*D,*e*D,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value (*n*) for each valid MSISDN, and is valid for (*e*)days. Cycle period is ignored.
PROM_16,*,*a*,*t*M,*e*D,1,IMM,*w*,*b*	Execute once on date (*a*). (*b*) bucket added to (*w*) wallet with value determined from MSISDN file for each valid MSISDN, and is valid for (*e*) days. Cycle period is ignored.

Chapter 7

Real-Time Notifications

Overview

Introduction

This chapter explains how the delivery of a real-time notification is initiated and what a real-time notification can contain.

For more information about real-time notifications and how you configure them, see the discussion on real-time notifications in *Charging Control Services User's Guide*.

In this chapter

This chapter contains the following topics.

Real-Time Notifications 367
Notification Construction 369

Real-Time Notifications

Wallet notification types

This table lists the events which will trigger a real-time wallet notification request.

Type of notification	Criteria
Charging	• Bucket value changes • Balance type matches balance changes • Value decreases • Old total balance value is strictly above threshold • New total balance is equal to or below threshold
Recharging	• Bucket value changes • Value increases • Old total balance value is strictly below threshold • New total balance value is equal to or above the threshold
Balance expiry	• Bucket expires • Balance type matches bucket expired • Old total balance value was strictly above threshold • New total balance value is equal to or below threshold
Wallet expiry	Wallet expires

Type of notification	Criteria
Wallet state change	• Wallet state changes • Old state different from new state • Old state matches notification old state field. (See note) • New state matches notification new state field. (See note) **Note:** If the notification field is configured as 'any state' (null), the compared wallet state (old or new) is considered to be the same.

For more information about configuring the different wallet notifications, see *Charging Control Services User's Guide*.

Additional SMS Notifications

An SMS notification can also be triggered when a real-time event occurs. The SMS notification is delivered as a SleeNotificationEvent through the notificationIF interface. It is sent to the destination MSISDN using the transport method defined in the SMS notification template. This will be one of the following:

- smsInterface (from SMSCIF)
- xmsTrigger (from MM)

Notes:

- SMS notification types and the associated message templates are configured in ACS, for further information see the *ACS Configuration* chapter in the *Advanced Control Services User's Guide*.
- For more information about smsInterface and notificationIF, see *Short Message Charging Bundle User's* and *Technical Guide*.
- For more information about xmsTrigger, see *Messaging Manager Technical Guide*.

DAP Notification Delivery

Each notification is delivered as a SleeDapXmlEvent event to the xmlIF interface. The name is configurable but if omitted will default to 'xmlIF'.

After a notification is sent, no check is made to verify that it was received.

Notification Export

Real-time notifications can be exported to external, custom software tailored to a user's specific requirement.

Scenario Notifications

If the VWS completes a successful voucher recharge using a scenario other than default, it will record the scenario ID in the voucher recharge EDR.

If you have configured real-time wallet notifications to provide recharge notifications, you must set up a notification template for each scenario.

The notification template to use is based upon the scenario provided in the notification request from the ccsCDRLoader plug-in. The scenario is not a variable part of the notification itself.

The notification templates must be named using this format:

`ACS.VOUCHER_TYPE SCENARIO`

Where:

- *VOUCHER_TYPE* is the name of the voucher type (from the **Name** field on the New or the Edit Voucher Type screen
- *SCENARIO* is the ID of the scenario from the ID column on the New or the Edit Voucher Type screen.

Example: If a subscriber recharges a voucher of Basic Recharge type, using Scenario 1, the template name should be:

```
ACS.Basic Recharge1
```

Example: When no scenario or the default scenario was used the template would be:

```
Your account has been recharged successfully with $2 Your new credit balance is $3
To check your balance(s), please dial *135#
```

When scenario 9 was used:

```
Your account has been recharged successfully using Power Charge Gold with $2 Your
new credit balance is $3 To check your balance(s), please dial *135#
```

Note: These templates are configured in addition to the existing SMS recharge template (ACS. AccountRecharge).

Notification Construction

Notification Templates

Notifications are constructed from a template. The template contains variables which are replaced with data supplied by the requesting process when the notification is constructed.

There are two main types of templates:

- ACS notifications. For more information, see *Advanced Control Services User's Guide*.
- DAP templates. For more information, see *Data Access Pack User's & Technical Guide*.

Fields Used in Notification Templates

A list of fields you can use in notification templates follows.

AMOUNT

The difference in the change to the balance. Used when there is a change to the balance value.

BALANCE_TYPE

The name of the balance type associated with this billing event. The BALANCE_TYPE field is delivered only for the charging and recharging notification types.

BALANCE_UNIT

The units of the balance associated with this billing event. The BALANCE_UNIT field is delivered only for the charging and recharging notification types.

CLI

The caller line identifier of the reference associated with this update. This field is delivered for all five notification types.

COST

The total cost associated with this billing event. The COST field delivers any one of the following three variable types.

Type	Format
Cash	Integer

Type	Format
Time	String in the format SS:HH; where SS is the seconds part and HH is the hundredths of seconds part.
Unit	Integer

Note: The COST field is delivered only for the charging notification type.

EXPIRED_AMOUNT

The expired amount associated with this billing event. The EXPIRED_AMOUNT field delivers any one of the following three variable types.

Type	Format
Cash	Integer
Time	String in the format SS:HH; where SS is the seconds part and HH is the hundredths of seconds part.
Unit	Integer

Note: The EXPIRED_AMOUNT field is delivered only for the balance expiry notification type.

NEW_BALANCE

The new total balance value of the balance associated with this billing event. The NEW-BALANCE field delivers any one of the following three variable types.

Type	Format
Cash	Integer
Time	String in the format SS:HH; where SS is the seconds part and HH is the hundredths of seconds part.
Unit	Integer

Note: The NEW-BALANCE field is delivered only for the charging and recharging notification types.

NEW_STATE

The new state of the wallet. The NEW_STATE field contains any one of the letters shown in this table.

Letter	Wallet State
A	Active
D	Dormant
F	Frozen
P	Pre-use
S	Suspended
T	Terminated

Note: The NEW_STATE field is delivered only for the wallet expiry and wallet state change notification types.

NOTIFICATION_NAME

The name of the notification being delivered. This field is delivered for all five notification types.

OLD_BALANCE

The old total balance value of the balance associated with this billing event. The OLD_BALANCE field delivers any one of the following three variable types.

Type	Format
Cash	Integer
Time	String in the format SS:HH; where SS is the seconds part and HH is the hundredths of seconds part.

Unit	Integer

Note: The OLD_BALANCE field is delivered only for the charging and recharging notification types.

OLD_STATE

The old state of the wallet. The OLD_STATE field contains any one of the letters shown in this table.

Letter	Wallet State
A	Active
D	Dormant
F	Frozen
P	Pre-use
S	Suspended
T	Terminated

Note: The OLD_STATE field is delivered only for the wallet expiry and wallet state change notification types.

PRODUCT_TYPE

The name of the product type associated with this wallet. This field is delivered for all five notification types.

RECHARGE_AMOUNT

The total recharge amount associated with this billing event. The RECHARGE_AMOUNT field delivers any one of the following three variable types.

Type	Format
Cash	Integer
Time	String in the format SS:HH; where SS is the seconds part and HH is the hundredths of seconds part.
Unit	Integer

Note: The RECHARGE_AMOUNT field is delivered only for the recharging notification type.

TIME_STAMP

The date and time the billing event was generated. This field is delivered for all five notification types.

WALLET_NAME

The name of the wallet type. Typical names are 'Business' or 'Personal'. This field is delivered for all five notification types.

Chapter 8

About Installation and Removal

Overview

Introduction

This chapter provides information about the installed components for the Oracle Communications Network Charging and Control (NCC) application described in this guide. It also lists the files installed by the application that you can check for, to ensure that the application installed successfully.

In this Chapter

This chapter contains the following topics.

Installation and Removal Overview 373
Checking the Installation 373

Installation and Removal Overview

Introduction

For information about the following requirements and tasks, see *Installation Guide*:

- NCC system requirements
- Pre-installation tasks
- Installing and removing NCC packages

CCS packages

An installation of Charging Control Services includes the following packages, on the:

- SMS:
  - ccsSms
  - ccsCluster (for clustered SMS
  - ccsDapSms
- SLC:
  - ccsScp
- VWS:
  - ccsBe
  - ccsVoucherBe

Checking the Installation

Introduction

Refer to these checklists to ensure that CCS has installed correctly.

The end of the package installation process specifies a script designed to check the installation just performed. They must be run from the command line.

Check SMS procedure

Follow these steps in this checklist to ensure CCS has been installed on an SMS machine correctly.

Step	Action
1	Log in to SMS machine as root.
2	Check the following directory structure exists with subdirectories: • **/IN/service_packages/CCS** • **/IN/html/Ccs_Service** • **/IN/html/Ccs_FeatureNodes**
3	Check the directories contain subdirectories and that all are owned by: • ccs_oper user (group esg)
4	Check for obvious errors in log files: • **/IN/service_packages/CCS/ccsSms.install.log** • **/IN/service_packages/CCS/ccsScp.install.log**
5	Log into the system as ccs_oper. **Note:** This step is to check that the ccs_oper user is valid.
6	Enter `sqlplus /` No password is required. **Note:** This step is to check that the ccs_oper user has valid access to the database.
7	Ensure that the required CCS tables have been added to the database. For a list of the tables which should have been added, see CCS database tables.
8	Check the entries of following file: **/etc/inittab** Inittab Entries Reserved for CCS on SMS: a. `ccs3 /IN/service_packages/CCS/bin/ccsBeOrbStartup.sh` (runs ccsBeOrb) b. `ccs4 /IN/service_packages/CCS/bin/ccsCDRLoaderStartup.sh` (runs ccsCDRLoader) c. `ccs5 /IN/service_packages/CCS/bin/ccsSSMDispatcherStartup.sh` (runs ccsSSMDispatcher) d. `ccs7 /IN/service_packages/CCS/bin/ccsCDRFileGeneratorStartup.sh` (runs ccsCDRFileGenerator) e. `ccs8 /IN/service_packages/CCS/bin/ccsProfileDaemonStartup.sh` (runs ccsProfileDaemon) f. `cc10 /IN/service_packages/CCS/bin/ccsChangeDaemonStartup.sh` (runs ccsChangeDaemon)
9	Check that the processes listed in the process lists are running on the relevant machine. For a list of the processes which should be running, see *Process list - SMS* (on page 376).
10	Tail logs for the processes listed in process list to ensure there are no errors.

Check SLC procedure

Follow these steps in this checklist to ensure CCS has been installed on an SLC machine correctly.

Step	Action
1	Log in to SLC machine as root.
2	Check the following directory structure exists with subdirectories: **/IN/service_packages/CCS**
3	Check the directory contains subdirectories and that all are owned by: • ccs_oper user (group oracle)
4	Log into the system as ccs_oper. **Note:** This step is to check that the ccs_oper user is valid.
5	Type `sqlplus /` No password is required. **Note:** This step is to check that the ccs_oper user has valid access to the database.
6	Ensure that the required CCS tables have been added to the database. For a list of the tables which should have been added, see CCS database tables - SCP.

Check VWS procedure

Follow the steps in this checklist to ensure CCS has been installed on a VWS machine correctly.

Step	Action
1	Log in to VWS machine as root.
2	Check the following directory structure exists with subdirectories: **/IN/service_packages/CCS**
3	Check the directory contains subdirectories and that all are owned by: • ccs_oper user (group esg)
4	Check for obvious errors in log file: **/IN/service_packages/CCS/ccsBe.install.log**
5	Log into the system as ccs_oper. **Note:** This step is to check that the ccs_oper user is valid.
6	Type `sqlplus /` No password is required. **Note:** This step is to check that the ccs_oper user has valid access to the database.
7	Ensure that the required CCS tables have been added to the database. For a list of the tables which should have been added, see CCS database tables.
8	Check the entries of the **/etc/inittab** file. Inittab Entries Reserved for CCS on VWS: a. `ccs8 /IN/service_packages/CCS/bin/updateLoaderWrapper.sh` (only used if smsExtras is installed to run an instance of updateLoader. For more information about updateLoader, see *Service Management System Technical Guide*) **Note**: To define a TZ that the NOTICE messages by updateLoader are logged in, add DEBUG_TZ environment variable in the **updateLoaderWrapper.sh** script before the exec statement. For example: `export DEBUG_TZ=Asia/Kolkata`

b. `ccs9 /IN/service_packages/CCS/bin/ccsMFileCompilerStartup.sh`
(runs ccsMFileCompiler)

c. `cc10 /IN/service_packages/CCS/bin/cmnPushFiles-ccsVWARSExpiry.sh`
(runs an instance of cmnPushFiles for ccsVWARSExpiry)

d. `cc11 /IN/service_packages/CCS/bin/cmnPushFiles-ccsRewards.sh`
(runs an instance of cmnPushFiles for ccsRewards)

e. `cc12 /IN/service_packages/CCS/bin/cmnPushFiles-ccsExpiryMsgs.sh`
(runs an instance of cmnPushFiles for ccsExpiryMessageGenerator)

f. `cc14 /IN/service_packages/CCS/bin/ccsChangeDaemon`
(runs ccsChangeDaemon)

9 Check that the processes listed in the process lists are running on the relevant machine. For a list of the processes which should be running, see *Process list - VWS* (on page 376).

Adding announcement sets automatically

NCC can provide a customized SQL script that adds an entire announcement set.

This script is run once at installation, from SMS as sms_oper.

If you wish to use this script then contact your Oracle account manager.

Process list - SMS

If the application is running correctly, the following processes should be running on each SMS, started from the inittab:

- ccsBeOrb
- ccsCDRLoader
- ccsSSMDispatcher
- ccsCDRFileGenerator
- ccsProfileDaemon

Process list - SLC

If the application is running correctly, the following processes should be running on each SLC, started during SLEE startup:

- BeClient
- ccsSSMMaster (runs on the SSMMaster SLC only)

Process list - VWS

If the application is running correctly, the following processes should be running on each VWS, started from the inittab:

- ccsMFileCompiler
- ccsChangeDaemon
- cmnPushFiles