Oracle® Communications Billing and Revenue Management

Developer's Guide

Release 15.1
F93230-01
April 2025

ORACLE

Oracle Communications Billing and Revenue Management Developer's Guide, Release 15.1

F93230-01

Copyright © 2017, 2025, 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, software documentation, data (as defined in the Federal Acquisition Regulation), 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," "commercial computer software documentation," or "limited rights data" 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®, Java, MySQL, and NetSuite 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

Preface

Audience xxxi
Documentation Accessibility xxxi
Diversity and Inclusion xxxi

Part I Introduction to Customizing BRM

1 About Customizing BRM

About Customizing BRM 1-1
Planning Your Customization 1-2
Guidelines for Customizing BRM 1-3
About the BRM Client Access Libraries 1-3
BRM Programming Tools 1-4

2 Implementation Defaults

Defaults for Offering Packages and Bundles to Customers 2-1
Defaults for Creating Customer Accounts 2-2
Defaults for Login Names and Passwords 2-3
Defaults for Validating Customer Account Creation Information 2-3
Defaults for Validating Payment Information 2-4
Defaults for Displaying and Sending Introductory Messages 2-6
Defaults for Billing 2-6
Defaults for Tax Calculation 2-8
Defaults for Payments and A/R 2-9
Defaults for Maintaining an Audit Trail of BRM Activity 2-11

3 Understanding Flists

About Flists 3-1
Opcode Input and Output Specifications 3-2
About Creating and Using an Flist 3-3

Adding Information to Flists 3-3
Removing Data (Pointers) from an Flist 3-4
Copying Data from an Flist 3-4
Destroying Flists 3-5
Flist Creation Samples 3-5
Using Compile-Time Flags to Avoid Errors in Flists 3-5
Flist Management Rules 3-7
Flist Field Memory Management Guidelines 3-8
Handling Errors 3-10
Modifying the Maximum Length of an Opcode Flist Field 3-10

4 Understanding Storable Classes

About Storable Classes and Objects 4-1
Storable Class Naming and Formatting Conventions 4-2
Subclassing 4-2
About Defining Storable Classes 4-2
Fields Common to All Storable Classes 4-3
Defining New Fields for Storable Classes 4-3
Reading Objects 4-4
Locking Objects when Reading them 4-4
Reading an Entire Object 4-4
Reading Fields in an Object 4-5
Creating Objects 4-6
PCM_OP_CREATE_OBJ Opcode Flags 4-6
Writing Fields in Objects 4-7
PCM_OP_WRITE_FLDS Opcode Flags 4-8
Incrementing Fields in Objects 4-8
Updating Decimal Data Types 4-9
Updating Integer Data Types 4-9
PCM_OP_INC_FLDS Opcode Flags 4-10
Deleting Objects 4-10
Deleting Fields in Objects 4-10
Managing a Large Number of Objects 4-11
Creating a Large Number of Objects 4-11
Editing a Large Number of Objects 4-11
Deleting a Large Number of Objects 4-12
Locking Objects when Editing or Deleting a Large Number of Objects 4-12
Improving Performance when Working with Objects 4-13
Locking Specific Objects 4-13
Disabling Granular Object Locking 4-16

5 Understanding the PCM API

About the PCM API 5-1
- Context Management Opcodes 5-2
- Base Opcodes 5-2
  - Search and Global Search Opcodes 5-2
- FM Opcodes 5-3
- About the PREP and VALID Opcodes 5-4
  - Validating Fields by Using Field Validation Editor 5-4

Header Files 5-4
About Opcode Usage 5-5
About Transaction Usage 5-5
- Transaction Handling: Required 5-5
- Transaction Handling: Requires New 5-5
- Transaction Handling: Supports 5-6

Calling PCM Opcodes 5-6
Manipulating Objects in Custom Applications 5-7
Supporting an Older Version of BRM 5-7

6 Accessing Configuration Files and Objects in Custom Code

Accessing pin.conf Files in Custom Code 6-1
Using /config/business_params Objects 6-2
- Adding and Loading New Parameters 6-2
- Adding and Loading New Parameter Classes 6-3
- Examples of Accessing Business Parameters in Custom Code 6-7
  - Calling Business Parameters from PCM_OP_PYMT_POL_VALIDATE_PAYMENT 6-8
  - Calling Business Parameters from PCM_OP_BILL_POL_REVERSE_PAYMENT 6-8

7 Understanding the BRM Data Types

About the BRM Data Types 7-1
Simple Data Types 7-2
Portal Object ID (POID) 7-2
Decimal Data Type 7-3
Arrays 7-4
Substructure 7-4
Buffer Data 7-5
- Setting Buffer Data Fields in an Flist 7-5
- Getting Buffer Fields From an Flist 7-6
- Specifying Buffer Data Fields in Flist Converted to Strings 7-6

Error Buffer 7-7

8 Using BRM SDK

About BRM SDK 8-1
About PCM SDK 8-1
BRM SDK Directory Contents 8-1
Deploying New and Customized Components 8-2
Deploying Applications 8-2
Deploying FMs 8-2
Deploying DMs 8-3
Compiling CMs for Purify 8-3

9 Finding Errors in Your Code

Detecting Errors in Your Code 9-1
Individual-Style ebuf 9-1
Series-Style ebuf 9-2
Error Handling Flow 9-2
Logging Errors and Messages 9-3
Diagnosing Application Problems 9-4
Detecting CM and DM Errors 9-4

10 Testing Custom Applications

Testing New or Customized Components 10-1
Testing Custom Applications 10-1
Testing New or Customized Policy FMs 10-1
Testing New or Customized DMs 10-2
Changing the Virtual System Time to Test BRM 10-2

11 Using the testnap Utility to Test BRM

About testnap 11-1
About Buffer Numbers 11-2
Executing Opcodes 11-2
Reading an Object and Fields 11-2
Reading Fields in an Object 11-2
Reading an Object and Writing Its Contents to a File 11-3
Retrieving Objects 11-3
Retrieving the Contents of the First Object Found 11-3
Retrieving the POID Field of the Objects Found 11-4

Creating a New Search Object 11-4
Retrieving Objects One at a Time 11-5
Retrieving a Specific Number of Objects at a Time 11-6
Creating Objects 11-7
Using a Text File to Create an Object 11-7
Using a Here Document to Create an Object 11-7
Manipulating External Buffer Fields 11-8
Reading Data in a Buffer to a File 11-8
Using Buffers to Concatenate Flists 11-9
Setting up Buffers and Displaying the List of Buffers 11-10
Creating and Displaying the Contents of a Buffer 11-10
Sorting an Flist 11-10
Invoking Shell Commands 11-11
Troubleshooting testnap 11-11
Error 27: Connection Error 11-12
Error 4: Login Failure 11-12
Incorrect Database Number 11-13
Error 26: DM Not Running 11-13
Invalid Buffer Index 11-14
Error 56: Failed to Connect 11-14

Part II Customizing BRM Server Components

12 About System and Policy Opcodes

Understanding System and Policy Facilities Modules 12-1
System FM Functions 12-1
Policy FM Functions 12-2
Policy Opcodes 12-2
Using the Policy Opcode Source Files 12-3
Using the Default Implementation with Your Custom Implementation 12-4
Adding a New Policy FM 12-4

13 Writing a Custom Facilities Module

About Implementing Custom FMs 13-1
Creating a New FM 13-1
Defining New Opcodes 13-2
Defining Input and Output Flist Specifications 13-3
Defining New Storable Class and Field Definitions 13-3
Writing a Function to Implement a New Opcode 13-3
Using the fm_post_init Function to Call Nonbase Opcodes at CM Initialization 13-4

Creating an Opcode-to-Function Mapping File 13-4
Creating a Shared Library for a New FM 13-6
About Configuring a New FM into a CM 13-6
Adding a New FM Module to the CM Configuration File 13-6
Initializing Objects for Multiple Processes 13-6
Handling Transactions in Custom FMs 13-7
Managing Memory in Custom FMs 13-8
Opening a New Context in an FM 13-8
Compiling and Linking a Custom FM 13-8
Configuring Your New Policy FM 13-9
Debugging FMs 13-9

14 Writing a Custom Data Manager

About Adding a Custom Data Manager 14-1
About Mapping Objects to Alternate Storage Mechanisms 14-1
About Adding Interfaces to Legacy Systems 14-1
Understanding the Data Manager Interface 14-2
Calling Conventions 14-2
Data Manager Memory Model 14-2
Function Entry Points 14-2
Argument Descriptions 14-3
Creating a Custom Data Manager 14-4
Creating a New Data Manager 14-4
Writing, Compiling, and Linking a Custom DM 14-5
Configuring Your Custom DM 14-5
Starting and Stopping Your Custom DM 14-5
Managing Memory 14-6
Handling Errors 14-6
Configuring Your CM to Use the Custom DM 14-6
Editing Your Custom Opcodes to Access the Custom DM 14-7

15 Creating Custom Fields and Storable Classes

Creating, Editing, and Deleting Fields and Storable Classes 15-1
Modifying the pin.conf File to Enable Changes 15-1
Increasing the Size of the CM Cache for the Data Dictionary 15-2
Using DDL when Updating the Data Dictionary Tables 15-2
Creating Custom Fields 15-2
Creating Custom Storable Classes 15-3
Making Custom Fields Available to Your Applications 15-4
About BRM SDK Opcodes 15-5

Using BRM SDK Opcodes to Manage Storable Classes 15-6
Creating and Modifying Storable Classes 15-6
Retrieving Storable Class Specifications 15-7
Deleting Storable Class Specifications 15-7
Using BRM SDK Opcodes to Manage Field Specifications 15-8
Creating and Modifying Field Specifications 15-8
Retrieving Field Specifications 15-8
Deleting Field Specifications 15-9
Converting Storable Class Files from Previous Versions 15-9
Deploying Custom Fields and Storable Class Definitions 15-9
Extracting Field and Storable Class Definitions with pin_deploy 15-10
Importing Storable Class Definitions with pin_deploy 15-11
Adding Fields to /config Objects 15-11
Using Developer Center to Modify /config Objects 15-12
Using testnap to Modify /config Objects 15-12

16 Storing Customer Profile Information

About Storing Customer Profile Information 16-1
Using Profile Objects to Collect Customer Profiles 16-1
Defining a Profile Subclass 16-1
Creating a Profile Object 16-1
Modifying a Profile Object 16-2
Deleting a Profile Object 16-2
Validating Profile Objects 16-2

17 Auditing Customer Data

Audit Trail Architecture 17-1
About Shadow Objects 17-2
Fields Marked for Auditing by Default 17-3
Enabling Auditing for a Field 17-3
Accessing Audit Trail Information 17-4
Using testnap to Retrieve Shadow Objects 17-5
Purging Archived Audit Data 17-5

18 Encrypting Data

About Encrypting Data 18-1
About AES Encryption 18-1
About Masking Data in Log Files 18-4
Encrypting Fields 18-4

Defining Masked Fields 18-4
About Encrypting Passwords 18-5
About the encryptpassword.pl Script 18-5
Encrypting Passwords Automatically for BRM Base Components 18-6
Encrypting Passwords Manually with OZT 18-7
Encrypting Passwords Manually with AES 18-7
Configuring the Data Manager for AES Encryption 18-8
Configuring the Data Manager for Oracle ZT PKI Encryption 18-8
Generating a Root Encryption Key 18-9
Modifying a Root Encryption Key 18-10

19 Searching for Objects in the BRM Database

About Searching for Objects 19-1
About the Search Input Flist 19-2
Search POID 19-3
Argument List 19-3
Results Array 19-3
Search Query 19-4
Flags 19-4
Search Query Syntax 19-5
About Searching for Objects by Their POID Subcomponent 19-6
Searching for Objects by the POID Database Number 19-7
Searching for Objects by the POID Type 19-7
Searching for Objects by the POID Object ID 19-7
Searching for Objects by the POID Revision Number 19-7
Search Query Syntax for Count-Only Searches 19-8
Search Query Syntax for Calculate-Only Searches 19-8
Using the PIN_FLD_PARAMETERS Field 19-9
Limiting Search Results by Using Row Numbers 19-10
Using the "in" Operator 19-10
Searching Subclasses 19-11
Returning Specific Storable Classes 19-12
Returning Entire Arrays 19-12
Search Template Examples 19-13
Using a Predefined Template 19-13
Defining the Search Template at Runtime 19-15
About Single-Schema Searches 19-16
Performing a Search on a Single Schema 19-17
Performing a Step Search on a Single Schema 19-18
Getting the Next Set of Search Results from a Step Search 19-19
Ending a Step Search 19-20

Simple Search Example 19-20
Step Search Example 19-21
Performing Exact Searches 19-24
Using "like" with Exact Searches 19-26
Exact Search Limitations 19-26
Complex Searches 19-27
Complex Search Example 19-28
About Performing Distinct Searches with Ordering and Pagination 19-28
Creating Storable Classes and Database Views for Distinct Searches 19-29
Performing Distinct Searches with Ordering and Pagination 19-29
Modifying Storable Classes for Distinct Searches 19-30
Search without POID 19-30
About Multischema (Global) Searches 19-31
Performing a Global Search 19-32
Performing a Global Step Search 19-33
Getting the Next Set of Search Results from a Global Step Search 19-34
Ending a Global Step Search 19-35
Global Search Example 19-35
Building the POID for the Input Flist 19-36
Building POID for the Input Flist in C 19-36
Building POID for the Input Flist in Java 19-36
The Impact of Searches on Shared Memory Allocation 19-37
Improving Search Performance 19-37
Removing Redundant Distinct Searches 19-37
Step Search Limits 19-38
Transaction Caching 19-38

20 Adding Support for a New Service

About Adding Support for a New Service 20-1
About BRM Services 20-1
About Supporting a New Service 20-1
Creating Service and Event Storable Classes 20-2
Setting Up Rating for a New Service 20-3
Setting Up Pricing Data for Online Rating 20-3
Mapping Event Types to a New Service Storable Class 20-3
Defining RUMs for New Service Usage Events 20-3
Setting Up Provisioning Tags for a New Service 20-3
Defining Impact Categories for a New Service 20-3
Defining Custom Balance Elements for a New Service 20-3
Specifying How to Round Balance Impacts for New Service Usage Events 20-4
Adding a New Service to Your Product Offerings 20-4

Configuring Sub-Balances to Track Specific Types of Usage for a New Service 20-5
Adding Database Partitions for New Service Usage Events 20-5
Loading Rated Events for a New Service into the Database 20-5
Setting Up Billing for a New Service 20-5
Setting Up Account Creation for a New Service 20-5
Setting Up Business Profiles for a New Service 20-5
Optional Support for a New Service 20-5
Synchronizing Data for a New Service with External Applications 20-6
Mapping Devices to a New Service 20-6
Providing Access to a New Service on the Web 20-6
Generating Usage Reports for a New Service 20-6

21 Using BRM Messaging Services

About the UMS Framework 21-1
Enabling Messaging 21-2
Creating and Loading Message Templates 21-3
Generating Messages in the Producer Application 21-3
Retrieving Message Templates 21-3
Retrieving Message Templates from /strings Objects 21-4
Creating Message Objects 21-4
Retrieving Message Objects in the Consumer Application 21-5

22 Using BRM with Oracle Application Integration Architecture

About Oracle Application Integration Architecture 22-1
Installing and Configuring the Required BRM Components 22-1
Integrating BRM Features with External CRM Applications 22-2
Integrating Collections with External CRM Applications 22-2
Integrating Friends and Family Promotions with External CRM Applications 22-3
Displaying Siebel CRM Promotion Names on Invoices 22-3
Integrating BRM Features with External CRM Applications in a Multischema System 22-3
Integrating Account Migrations with External Applications in a Multischema System 22-4
Integrating Collections with External Applications in a Multischema System 22-5
Creating Charge Offers and Discount Offers for an External CRM 22-5
Creating Charge Offers with Different Prices for Multiple Price Lists 22-5
Validating Customer Contact Information 22-5

23 Using Event Notification

About Event Notification 23-1
About the Event Notification List 23-1

Implementing Event Notification 23-2
Merging Event Notification Lists 23-2
Editing the Event Notification List 23-3
Triggering Multiple Opcodes with One Event 23-4
Triggering Custom Operations 23-4
Loading the Event Notification List 23-5
About Notification Events 23-6

24 Writing Custom Batch Handlers

About Batch Handlers 24-1
Configuration Parameters 24-1
Batch Handler Work Flow 24-1

25 Managing Devices with BRM

About the Device Management Framework 25-1
Implementing Device Management 25-1
Creating Devices 25-2
Managing the Device Life Cycle 25-2
Managing Device Attributes 25-3
Associating Devices and Services 25-3
Deleting Devices 25-3
Tracking Device History 25-4
Device Management and Multischema Environments 25-4
Configuring Event Notification for Device Management 25-4
Defining the Device Life Cycle 25-5
Localizing Device State Names 25-6
Customizing Device State Changes 25-7
Defining Device-to-Service Associations 25-7
Creating Custom Device Management Systems 25-8

26 Managing Orders

About Order Manager 26-1
Implementing Order Manager 26-1
Creating Orders 26-2
Processing Order Response Files 26-2
Managing the Order Life Cycle 26-2
Associating or Disassociating Orders with Master Orders 26-3
Managing Order Attributes 26-3
Deleting Orders 26-3

Tracking Order History 26-4
Managing the Order History Log 26-4
Installing Order Manager 26-4
Uninstalling Order Manager 26-5
Order Management and Multischema Environments 26-5
About Defining the Order Life Cycle 26-5
Creating Custom Order Management Systems 26-7
About the Order Management Opcodes 26-7
Creating /order Objects 26-8
Customizing Order Creation 26-8
Processing Order Response Files 26-9
Customizing Order Processing 26-9
Associating and Disassociating /order Objects 26-9
Customize How to Validate Association and Disassociation 26-10
Updating /order Objects 26-10
Setting the State in /order Objects 26-10
Changing /order Object Attributes 26-11
Deleting /order Objects 26-12
Customizing How to Delete Orders 26-12

Part III Integrating BRM with Enterprise Applications

27 About Enterprise Application Integration (EAI) Manager

About Integrating BRM with Enterprise Applications 27-1

28 Installing EAI Manager

About Installing EAI Manager 28-1
Software Requirements 28-1
Installing EAI Manager 28-1
Increasing Heap Size to Avoid "Out of Memory" Error Messages 28-2
Configuring Event Notification for EAI Manager 28-3

29 Payload Configuration File Syntax

About the Payload Configuration File Syntax 29-1
Publisher Definitions 29-2
Event Definitions 29-3
Element Definitions 29-5
Syntax of Elements and Attributes 29-6
Source 29-6

Tag 29-6
StartEvent 29-7
EndEvent 29-7
DataFrom 29-7
UseOnlyElement 29-7
UseElementId 29-8
Attribute 29-8
DTD 29-8
PinFld 29-9
Field 29-9
ExtendedInfo 29-10
Search 29-10
SubElement 29-11
PcmOpFlag 29-11
ReadFlds 29-11
ReadObj 29-12
ElemId 29-12
ReadValFrom 29-12
Event Flist, Event Definition, and XML Output Example 29-13

30 Filtering which Business Events Are Published

Filtering which Business Events Are Published 30-1
About the Condition Attribute 30-1
About the Condition Definition 30-2

31 Building a Connector Application

Building a Connector Application 31-1

32 Configuring EAI Manager

Configuring the Connection Manager for EAI 32-1
Configuring the EAI DM 32-2
Configuring the Payload Generator EM 32-3
Specifying the Date and Time Format for Business Events 32-4
Defining Infinite Start Date and End Date Values 32-4
Configuring EAI Manager to Publish to an HTTP Port 32-5

33 Configuring Business Events

About BRM Business Events 33-1

About Publishing Additional Business Events 33-2
Setting Up Multiple Publishers and Events 33-2
Defining Business Events 33-2
Removing Events That You Do Not Want to Publish 33-4
Returning Identifiers from Enterprise Applications 33-4
Changing the Format of Published Events 33-4
Validating Your Changes to the Payload Configuration File 33-5

34 EAI DM Functions

AbortTransaction 34-1
CommitTransaction 34-1
FreeGlobalContext 34-2
GetGlobalContext 34-2
Initialize 34-3
OpenTransaction 34-3
PrepareCommit 34-4
PublishEvent 34-4
SetIdentifier 34-5
Shutdown 34-5

Part IV Integrating BRM with an Apache Kafka Server

35 About Integrating BRM with an Apache Kafka Server

About Integrating BRM with Kafka Servers 35-1
About the EAI Framework for the Kafka DM 35-3
About the CM and Notification Events 35-3
About the Kafka DM 35-4

36 Configuring BRM to Publish Notifications to Kafka Servers

Overview of BRM Configuration Tasks for Kafka Servers 36-1
Installing the BRM Kafka DM 36-2
Configuring Thread Pooling for the Kafka DM 36-2
Enabling SSL between Kafka DM and Kafka Server 36-3
Configuring Event Notification for Kafka Servers 36-4
Defining Business Events for Your Kafka Server 36-5
Mapping Business Events to Kafka Topics 36-6
About Setting Topic and Payload Keys 36-6
Adding Headers to Messages 36-7
Adding Separate Payload Settings 36-8

Mapping Flist Fields to Payload Tags 36-9
Editing the dm_kafka_config.xml File 36-11
Configuring the Dynamic Key Value 36-13
Configuring Where to Record Failed Events 36-14
Customizing Notification Enrichment 36-15

Part V Creating and Customizing Client Applications

37 Adding New Client Applications

About Adding New Client Applications 37-1
Using Existing System Opcodes 37-1
Using Custom Opcodes 37-2
Using a Custom Data Manager (DM) 37-2
Implementing Timeout for Requests in Your Application 37-3
Configuring Your Custom Application 37-3
Creating a Client Application in C 37-4
Compiling and Linking Your Programs 37-5
Guidelines for Developing Applications in C on Linux Platforms 37-5
Using the Sample Applications 37-5
Sample Applications 37-5
Policy FM Source Files 37-6
About Adding Virtual Column Support to Your Applications 37-6

38 Using Transactions in Your Client Application

Using Transactions 38-1
Types of Transactions 38-2
Read-Only Transactions 38-2
Read-Write Transactions 38-2
Transaction with a Locked Objects 38-2
Transaction with a Locked Default Balance Group 38-3
About Committing Transactions 38-3
About Cancelling Transactions 38-4
About the Transaction Base Opcodes 38-4
Customizing How to Open Transactions 38-4
Customizing the Verification Process for Committing a Transaction Opcode 38-4
Customizing How to Commit a Transaction 38-5
Customizing How to Cancel Transactions 38-5

39 Adding or Changing Login Options

About Customizing the Login Account for Your Application 39-1
- Creating an Account for Your Application 39-1
- Providing Login and Password to Your Custom Application 39-1
- Configuring System Passwords 39-2

Creating Several admin_client Services with Different Permissions 39-2

40 Creating Applications that Run on Multischema Systems

About Working with Multiple Schemas 40-1
Creating Accounts in a Multischema System 40-1
Maintaining Transactional Integrity 40-2
Searching for Accounts across Database Schemas 40-2
Finding How Many Database Schemas You Have 40-3
Bill Numbering 40-3

41 Creating BRM Client Applications by Using the MTA Framework

About the BRM MTA Framework 41-1
- BRM MTA Framework Layers 41-2
- MTA Stages 41-3
  - MTA_CONFIG Execution Stage 41-4
  - MTA_INIT_APP Execution Stage 41-4
  - MTA_INIT_SEARCH Execution Stage 41-4
  - Search Execution 41-4
  - MTA_TUNE Execution Stage 41-5
  - Job Distribution 41-5
  - MTA_JOB_DONE Execution Stage 41-5
  - MTA_EXIT Execution Stage 41-5
  - MTA_WORKER_INIT Execution Stage 41-5
  - MTA_WORKER_JOB Execution Stage 41-6
  - Worker Thread Job Execution 41-6
  - MTA_WORKER_JOB_DONE Execution Stage 41-6
  - MTA_WORKER_EXIT Execution Stage 41-6
- MTA Global Flist Structure 41-6

Using the BRM MTA Framework 41-8
- MTA Callback Functions 41-9
- MTA Helper Functions 41-10
- MTA Policy Opcode Hooks 41-11

Creating a Multithreaded BRM Client Application 41-12
- Searching Different Data Sources 41-12

Displaying Application Help Information 41-13
Error Notifications 41-15
Customizing BRM Multithreaded Client Applications 41-15
Implementing the MTA Policy Opcodes 41-16
Configuring the MTA Policy Opcodes 41-16
Configuring your Multithreaded Application 41-17
Applying Configuration Entries to Specific Utilities 41-18
Using Multithreaded Applications with Multiple Database Schemas 41-18
MTA Policy Opcode Hooks 41-18
MTA_CONFIG 41-18
MTA_ERROR 41-19
MTA_EXIT 41-19
MTA_INIT_APP 41-19
MTA_INIT_SEARCH 41-20
MTA_JOB_DONE 41-20
MTA_TUNE 41-21
MTA_USAGE 41-21
MTA_WORKER_EXIT 41-21
MTA_WORKER_INIT 41-22
MTA_WORKER_JOB 41-22
MTA_WORKER_JOB_DONE 41-22
MTA Callback and Helper Functions 41-23
pin_mta_config 41-23
pin_mta_exit 41-24
pin_mta_get_decimal_from_pinconf 41-24
pin_mta_get_int_from_pinconf 41-25
pin_mta_get_str_from_pinconf 41-26
pin_mta_global_flist_node_get_no_lock 41-26
pin_mta_global_flist_node_get_with_lock 41-27
pin_mta_global_flist_node_put 41-27
pin_mta_global_flist_node_release 41-28
pin_mta_global_flist_node_set 41-28
pin_mta_init_app 41-29
pin_mta_init_search 41-29
pin_mta_job_done 41-30
pin_mta_main_thread_pcm_context_get 41-31
pin_mta_post_config 41-31
pin_mta_post_exit 41-32
pin_mta_post_init_app 41-32
pin_mta_post_init_search 41-32
pin_mta_post_job_done 41-33
pin_mta_post_tune 41-33

pin_mta_post_usage 41-34
pin_mta_post_worker_exit 41-34
pin_mta_post_worker_init 41-35
pin_mta_post_worker_job 41-35
pin_mta_post_worker_job_done 41-36
pin_mta_tune 41-37
pin_mta_usage 41-37
pin_mta_worker_exit 41-38
pin_mta_worker_init 41-38
pin_mta_worker_job 41-38
pin_mta_worker_job_done 41-39
pin_mta_worker_opcode 41-40

42 Creating Client Applications by Using Java PCM

About Using the Java PCM API 42-1
Software Requirements 42-2
About the Java PCM API and the C API 42-2
Using the Java PCM API 42-2
About Creating Client Applications by Using the Java PCM API 42-3
About Synchronous and Asynchronous Modes 42-3
Actions Performed by BRM Java Client Applications 42-3
Opening a PCM connection 42-3
Using Custom Fields in Java Applications 42-4
Creating Custom Storable Classes 42-4
Calling Custom Opcodes 42-5
Using Synchronous Mode for Opcode Processing 42-5
Getting a Text Format of an Flist 42-5
Handling Exceptions 42-5
Logging Errors and Messages 42-6
Specifying a Timeout Value for Requests 42-6
Using the Asynchronous PCP Mode in Java PCM Client Libraries 42-7
About PCPSelector 42-7
About PortalContextListener 42-8
How Asynchronous Mode for Opcode Processing Works 42-8
Creating Client Applications for Asynchronous Mode of Opcode Processing 42-9
Setting Global Options 42-10
Default Entries in the Infranet.properties File 42-10
Optional Entries in the Infranet.properties File 42-11
Example Infranet.properties File 42-13
Controlling Opcode Logging from a Client Application 42-13
Running the jnap Utility 42-14

Getting Help with jnap 42-14
Example of Using jnap 42-15
About the Sample Program 42-15

43 Creating Client Applications by Using Perl PCM

About the Perl API 43-1
Differences between the Perl API and the C API 43-1
Guidelines for Using the Pcmif Module 43-2
Performing PCM Operations 43-2

44 Creating Client Applications by Using PCM C++

About PCM C++ 44-1
Skills Required 44-1
Installation 44-2
Comparison of the PCM C++ and PCM C APIs 44-2
Understanding PCM C++ Concepts 44-5
Passing Arguments 44-5
Using Arrays 44-6
Using Smart Pointers to Manage Memory 44-6
Construction and Destruction 44-7
Copying and Assignment 44-7
Using Field Value Ownership 44-8
Using PinBigDecimal 44-9
Field Value Ownership 44-9
Using PinBigDecimal with Flists 44-9
Using the toString() Method 44-10
Using the Divide Method 44-10
Using a Null Pointer 44-11
Handling Exceptions 44-11
Logging to pinlog 44-13
Accessing Configuration Values by Using pin.conf 44-13
Using PCM C++ with PCM C 44-14
Using the PCM C++ API 44-14
Opening a PCM Connection 44-14
Closing a PCM Connection 44-16
Creating Custom Fields 44-16
Creating an Flist 44-17
Getting an Flist in Text Format 44-17
Debugging PCM C++ Programs 44-17

Troubleshooting 44-18

Part VI Customizing Customer Center and Self-Care Manager

45 Using Customer Center SDK

About Customer Center SDK 45-1
About Using Customer Center SDK to Customize Customer Center 45-1
About Using Customer Center SDK to Customize Self-Care Manager 45-2
Contents of Customer Center SDK 45-2
Customer Care API Reference 45-3
Coding Your Customizations 45-3
About Compiling and Packaging Your Customizations 45-3
Coding, Building, and Deploying Customizations 45-5
Syntax for the buildAll Script 45-5
Syntax 45-5
File Location 45-5
Parameters 45-5
Testing Your Customizations for Customer Center 45-5
Understanding the BRM Business Application SDK Framework 45-6
The Model-View-Controller Architecture 45-6
How the Controllers Work 45-7
Example Data Flow between a Simple Field and BRM 45-7
About Field Components 45-7
Displaying Versus Saving Data in Fields 45-8
Display Fields and Controllers 45-9
About PModelHandle 45-9
About Lightweight Components (Self-Care Manager Only) 45-10
Source Code Examples 45-10

46 Customizing the Self-Care Manager Interface

About Customizing Self-Care Manager 46-1
Hardware and Software 46-1
Understanding Self-Care Manager Components 46-1
About PInfranetServlet 46-2
Using PInfranetServlet to Process Requests 46-2
Example Data Flow Designs 46-3
Design 1 46-4
Design 2 46-5
Design 3 46-6
Extending the Functionality of Self-Care Manager 46-7

Adding Fields 46-8
Removing Fields 46-9
Creating a New Component 46-9
Creating a Link for the JSP Pages for a Get Request 46-10
Creating a Link for the JSP Pages for a Post Request 46-10
Designing a Component 46-10
Developing the Customizable Component 46-11
Developing a Noncustomizable Component 46-15
Error Handling 46-16
Formatting Your Data 46-17
Method 1: Add Java Code to Your JSP Pages 46-17
Method 2: Use a Formatting Bean that Contains the Presentation Logic for the Data. 46-17
Building the Self-Care Manager Components 46-18
Self-Care Manager Customization Examples 46-18

47 Customizing the Customer Center Interface

Customizing and Configuring Customer Center 47-1
Tools and Techniques for Customizing Customer Center 47-2
Customization Procedure Overview 47-2
Coding Your Customizations 47-2
Building and Deploying Your Customizations 47-3
Modifying the Customer Center Properties Files 47-3
About the Default Customer Center properties Files 47-3
Modifying Behaviors Defined by the Default Properties Files 47-3
Displaying Event Timestamps with Seconds Precision 47-4
Adding Inactive Product Status Indicators 47-5
Changing the List of Services Available in the Search Panel 47-5
Improving Account Search Performance 47-5
Changing Number Searches for GSM Services 47-6
Modifying the Shortcut Key Sequences 47-6
Specifying the Number of Bills Displayed in the Balances Tab 47-7
Suppressing the "Missing Login/ID" Message for Custom Service Panels 47-7
Changing the Maximum Number of Security Code Characters 47-7
Updating Notes Before Saving 47-8
Reminding CSRs to Customize Deals Before Completing a Purchase 47-8
Identifying Services by Device ID Rather than Login ID 47-8
Adding a Tax Exemption Type 47-9
Customizing Event Searches 47-9
Customizing the Case Sensitivity of Event Searches 47-9
Customizing the Selections for Service Type in Event Searches 47-10
Customizing the Selections for Service Status in Event Searches 47-10

Customizing the Selections for Device Type in Event Searches 47-11
Customizing Balance Group Searches 47-11
Customizing the Case Sensitivity of Balance Group Searches 47-12
Customizing the Selections for Service Type in Balance Group Searches 47-12
Customizing the Selections for Service Status in Balance Group Searches 47-13
Customizing the Selections for Device Type in Balance Group Searches 47-13
Customizing Product/Discount Searches 47-14
Customizing the Case Sensitivity of Product/Discount Searches 47-14
Customizing the Selections for Service Type in Product/Discount Searches 47-15
Customizing the Selections for Service Status in Product/Discount Searches 47-15
Customizing the Selections for Device Type in Product/Discount Searches 47-16
Customizing Service Searches 47-17
Customizing the Case Sensitivity of Service Searches 47-17
Customizing the Step Search Size 47-17
Customizing the Selections for Service Type in Service Searches 47-18
Customizing the Selections for Service Status in Service Searches 47-18
Customizing the Selections for Device Type in Service Searches 47-19
Hiding the Password Fields in Customer Center 47-19
Disabling the Child Amounts Check Box 47-20
Adding a Custom Service as a Login to Customer Center 47-20
Using Customer Center SDK Scripts for Customer Center 47-20
Adding New Pages to the Customer Center Interface 47-21
About Portal Infranet Aware Widgets 47-22
Adding Account Maintenance Pages 47-22
Overview of Account Maintenance Components 47-22
Saving Changes 47-23
Refreshing Data in the UI 47-24
Currency Toggling 47-25
Drill-Down Links 47-26
Advanced Drill-Down Techniques 47-28
Modifying the Customer Center Permissions 47-29
Adding Your Page to the Customer Center Toolbar 47-29
Adding New Account Wizard Pages 47-29
Understanding the New Accounts Wizard 47-30
Base Storable Classes for Account Creation Pages 47-31
Methods Used in New Account Creation Pages 47-31
Removing a Payment Method from Customer Center 47-32
Advanced Customer Center Concepts 47-34
Components Used in Customer Center 47-35
Portal Infranet-Aware Components 47-35
Graphical Components 47-35
Nongraphical Components 47-37

Data Manager Components 47-38
About the BAS Data Flow with a Swing-Compatible UI 47-39
About Field Specifications 47-39
About Controller Processing 47-40
Building Your Customer Center Customizations 47-40
Creating a Self-Signed Java Security Certificate 47-40
Modifying the signjar Script 47-41
Building Your Customization Files 47-41
Requirements 47-41
Using the buildAll Script 47-41
Testing Your Customizations 47-42
Deploying Your Customer Center Customizations 47-43
Deploying Customer Center Customizations on Linux 47-43
About the Customer Center Properties Files 47-43
Default Properties Files 47-44
Configurator Properties Files 47-44
Customized Properties Files 47-45
Other Properties Files 47-45
Deploying Customer Center Customizations on Windows 47-46
Customer Center Customization Examples 47-46

48 Using Configurator to Configure Customer Center

About Configurator 48-1
Using Configurator 48-1
What's Next? 48-2
Configuring Customer Center Account Maintenance Pages 48-2
Using the Account Maintenance Configurator Tabs 48-3
Summary Configurator 48-3
Modifying the Customer Type List 48-4
Contacts Configurator 48-5
Balance Configurator 48-6
Payment Configurator 48-7
Plan Configurator 48-8
Service Configurator 48-8
Hierarchy Configurator 48-9
Sponsorship Configurator 48-11
Sharing Configurator 48-11
Other Settings 48-12
Account Search Results Configurator 48-13
Starting Account Search Configurator 48-13
Adding a New Search Criteria field 48-14

Modifying a Search Criteria Field 48-17
Deleting a Custom Search Criteria Field 48-17
Tab Options 48-17
Reordering Pages 48-18
Modifying Attributes of an Existing Page 48-18
Hiding an Existing Page 48-18
Adding a New Page 48-18
Removing a Custom Page 48-18
Configuring the Customer Center New Accounts Wizard 48-19
Contacts Panel 48-19
General Panel 48-20
Payment Panel 48-21
Billing Panel 48-22
New Account Page Options 48-22
Reordering New Account Pages 48-22
Modifying an Existing Page 48-23
Hiding an Existing Page 48-23
Adding a New Page 48-23
Removing a Custom Page 48-23
Using the Configurator Resource String Editor 48-24
Starting the Resource String Editor 48-24
Searching for Labels to Replace 48-24
Resource String Editor String Search Rules 48-24
Replacing Labels with New Strings 48-24
Undoing Label Changes 48-25
Additional Configured Profile Panel Examples 48-25

49 Adding Custom Fields to Customer Center

Coding and Deploying Custom Fields for Customer Center 49-1
Adding Custom Fields to Infranet.properties 49-1
Generating Your Custom Field Java Source Code 49-2
Compiling and Signing Your Custom Fields Java Source Code 49-3
What's Next 49-4
Configuring JBuilder to Add Custom Fields to Customer Center 49-4
What's Next 49-7
Building and Deploying Your New Profile Panel 49-7

50 Setting Up JBuilder to Customize the Customer Center Interface

About Using JBuilder to Customize the Customer Center Interface 50-1
Adding PIA Widgets to the JBuilder Palette 50-1

Creating a JBuilder Project for Customer Center SDK 50-1

51 Creating a New Customer Center Service Panel

Creating a New Service Panel 51-1
Correcting Field Alignment 51-8
What's Next 51-9

52 Creating a New Customer Center Profile Panel

Creating a New Profile Panel 52-1
What's Next 52-8

53 Sample Customer Center Customizations

Building and Deploying Customizations 53-1
Customizing Contact Fields 53-1
Customizing Contact Fields 53-1
Adding Drop-Down Lists to the Contact Type and Salutation Fields 53-1
Populating Drop-Down List Values from a Properties File 53-2
Adding Drop-Down Lists to Address Panel Fields 53-2
Adding and Removing Item Listeners to Address Field Drop-Down Lists 53-3
Modifying Multiple Contact Behavior 53-4
Specifying the Contact Type for Each Consecutive Contact 53-4
Disabling Changes to the Contact Type for the First Contact 53-4
Configuring Duplicate Checking for the Contact Type Field 53-5
Using Custom Address Panel and Contact Page 53-5
Replacing the Address Panel with a Custom Panel 53-6
Replacing the Contact Page with a Custom Page 53-6
Customizing Fields in the Balance Tab 53-6
Setting the Correct JRadioButtonMenuItem Button 53-6
Customizing Fields in the Payments Tab 53-7
Disabling the Billing Cycle & Tax Setup Link in the Payments tab 53-7
Configuring Values in the Billing Day of Month Combo Box 53-7
Setting the Next Billing Cycle Field to Visible or Not Visible 53-8
Customizing the Expiration Date Fields in the Credit Card Panel 53-8
Creating a Custom Payment Method 53-9
Customizing Fields in the Services Tab 53-10
Adding Charges for SIM and MSISDN Changes 53-11
Adding a Secondary MSISDN for Supplementary Services 53-11
Customizing Fields in the Hierarchy Tab 53-12
Adding a Custom Popup Component to the No Hierarchy Page 53-12

Adding a Custom NoHierarchy Page 53-12
Creating Customized Search Dialogs and Disabling the To Field 53-13
Adding Custom Options to the Actions Drop-Down Lists 53-13
Customizing Fields in the Sharing Tab 53-14
Adding a New Sharing Type to the View Drop-Down List 53-14
Configuring Dynamic Drop-Down Lists 53-15

Part VII Localizing BRM

54 Using BRM in International Markets

Supporting Multiple Currencies 54-1
Accepting Credit Card Payments in Multiple Currencies 54-1
Supporting Multiple Languages 54-2
Using Localized Client Applications 54-2
Localizing BRM 54-2

55 BRM Internationalization and Localization

About Localizing and Internationalizing 55-1
About Internationalization of BRM Client Applications 55-1
Writing Localized MFC Client Applications 55-2
About Internationalized Development on BRM 55-2

56 Creating a Localized Version of BRM

About the Localization SDK 56-1
Localization SDK Contents 56-2
Java Client Applications 56-2
Self-Care Manager Server Application 56-2
BRM Server Files 56-2
Localizations Supported 56-3
System Requirements for the Localization SDK 56-3
Building the Clients 56-4
Changing the Currency Format 56-4
Building Properties Files 56-4
Preparing Customer Center 56-6
Packaging Your BRM Client Localizations 56-6
Modifying Localized Versions of Customer Center 56-7
About Simple Customization 56-7
About Advanced Customization 56-8
Before You Begin 56-8

Simple Customization for Localized Versions of Customer Center 56-8
Deploying a Simple Customization of Customer Center 56-9
Advanced Customization for Localized Versions of Customer Center 56-9
Deploying an Advanced Customization of Customer Center 56-11
When to Use the Localization SDK for Advanced Customization 56-11
Localizing Self-Care Manager 56-12
Translating the Self-Care Manager Localized Strings File 56-13
Creating a Localized Self-Care Manager Installation for Linux 56-13
Localizing and Customizing Strings 56-14
Creating New Strings and Customizing Existing Strings 56-15
Localizing Existing Strings 56-16
Loading Localized or Customized Strings 56-16
Localizing BRM Reports 56-17
About Customizing Server Software 56-17
Setting the Default Language for Customer Accounts 56-17
Customizing Canonicalization 56-18
Exporting Data to an LDAP Server 56-18
Locale Names 56-18

57 Handling Non-ASCII Code on the BRM Server

About Character-Encoding Conversion 57-1
About Converting Multibyte or Unicode to and from UTF8 57-1
Direct Conversion Macros 57-2
Supporting Functions and Macros 57-3
Universal Macros 57-3
PIN_CONVERT_MBCS_TO_UTF8 57-3
PIN_CONVERT_STR_TO_UTF8 57-5
PIN_CONVERT_UNICODE_TO_UTF8 57-5
PIN_CONVERT_UTF8_TO_MBCS 57-6
PIN_CONVERT_UTF8_TO_STR 57-8
PIN_CONVERT_UTF8_TO_UNICODE 57-9
pin_IsValidUtf8 57-10
PIN_MBSLEN 57-11
PIN_SETLOCALE 57-12
Conversion Code Example 57-13

Part VIII Programming Utilities

58 Developer Utilities

load_config 58-1
load_config_provisioning_tags 58-4
load_localized_strings 58-6
load_pin_config_business_type 58-7
load_pin_device_permit_map 58-9
load_pin_device_state 58-10
load_pin_excluded_logins 58-13
load_pin_order_state 58-14
parse_custom_ops_fields 58-16
pin_adu_validate 58-17
pin_bus_params 58-19
pin_cfg_bpdump 58-21
pin_crypt_app 58-22
pin_deploy 58-24
pin_uei_deploy 58-27
pin_virtual_time 58-29
testnap 58-32
pin_config_editor 58-36
generate_config_xml_from_opcode_spec.py 58-37

Preface

This guide describes how to extend and customize Oracle Communications Billing and Revenue Management (BRM).

Audience

This guide is intended for developers.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Part I

Introduction to Customizing BRM

This part provides an overview of customizing Oracle Communications Billing and Revenue Management (BRM). It contains the following chapters:

- About Customizing BRM
- Implementation Defaults
- Understanding Flists
- Understanding Storable Classes
- Understanding the PCM API
- Accessing Configuration Files and Objects in Custom Code
- Understanding the BRM Data Types
- Using BRM SDK
- Finding Errors in Your Code
- Testing Custom Applications
- Using the testnap Utility to Test BRM

1

About Customizing BRM

Learn about customizing Oracle Communications Billing and Revenue Management (BRM) by using customized BRM code and by creating custom applications.

Topics in this document:

- About Customizing BRM
- Planning Your Customization
- Guidelines for Customizing BRM
- About the BRM Client Access Libraries
- BRM Programming Tools

Before customizing BRM, read *BRM Concepts*.

> **Caution:**
>
> - Always use the BRM API to manipulate data. Changing data in the database without using the API can corrupt the data.
> - Do not use SQL commands to change data in the database. Always use the API.

About Customizing BRM

There are different levels at which you can customize BRM:

- You can write custom client applications.
- You can change the default policy opcodes or write new policy opcodes.
- You can create a new storable class to hold information needed for your business. For example, if you provide a new type of service, such as online gaming, you must create a storable subclass of type ***/service*** to hold gaming-related information.
- You can write a new DM to access data in a storage system.

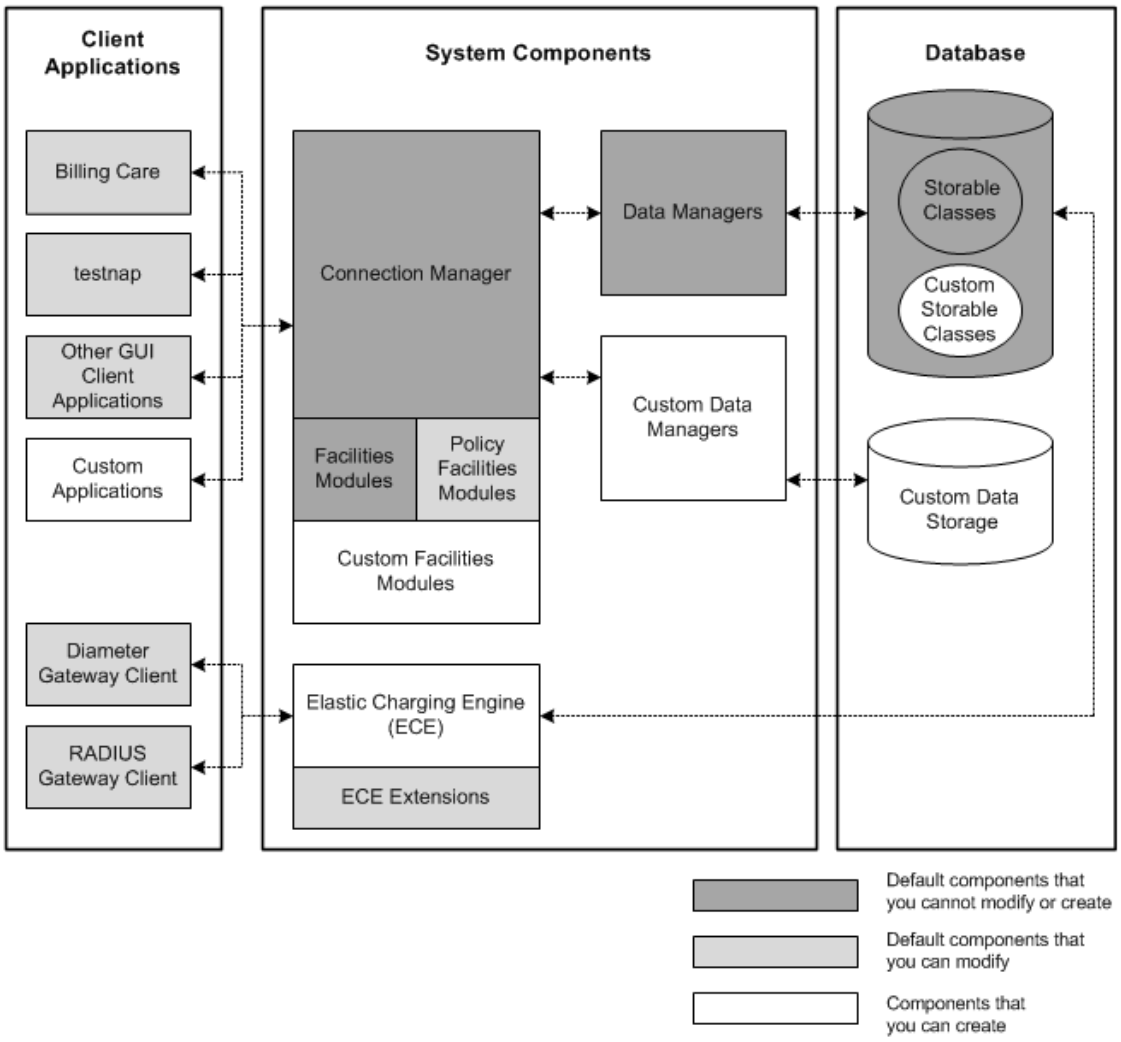
When you integrate your services, you can:

- **Create new services.** BRM already supports a basic set of services, but you can integrate your own services such as fax, disk storage, voice mail, and different types of broadband access such as DSL.
- **Customize existing services** to capture additional data. For example, you can extend the existing broadband service to charge different amounts for different types of access.
- **Customize how you charge for services** (for example, by time or quantity).

For more information, see "Adding Support for a New Service".

Figure 1-1 shows the different types of customization.

Figure 1-1 Types of Customization



Planning Your Customization

Before you change the default BRM implementation or add new features and functionality to BRM, you must plan your customization. For your customization to work properly, changes you make to one component might require changes to other components. For example, to support a new service such as online gaming, you must consider making the following additions or changes to BRM:

- Create a service class to define the service and to define events associated with the service that you want to rate.
- Develop product offerings to define charges for the service.
- Define balance elements for events generated by the service, and include them in your product offerings.
- Map attributes of the events to charges and impact categories by using the charge selector in PDC.
- Define different configurations for the service and rate each configuration by using charge-offer provisioning.

- Set up authentication and authorization for users to log in to the service.
- Extend client applications, such as Billing Care, and the account creation interface to process information specific to the service and to display information related to the service.
- Customize the default policy FMs to process information specific to the service, or write new system FMs and policy FMs if necessary.

In addition, depending on the level of customization, you must determine whether you can implement your changes by using a client tool, by using a configuration file, or by programming.

Guidelines for Customizing BRM

Here are some general guidelines that you should follow when customizing BRM:

- **Performance:** Assess how the proposed solution will impact the performance of the system. Review the sizing of the system to comply with existing performance requirements. The proposed solution should limit the number of charge offers, balance elements, and extended rating attributes (ERAs) affected during the customization.
- **Manageability:** Verify that the proposed solution is easy to maintain. Minimize customizations for the following reasons:
  - Incompatibilities might arise between customizations and product upgrades, making the move to a newer version difficult.
  - Incompatibilities might arise between customizations and patches, leading to side effects or the need to check and possibly change the customization whenever a new patch is released.

  Before implementing the customization, save the copies of unedited configuration files, properties files, policy code, or any other files changed in the customization. Include comments explaining the customizations in the policy code or configuration file.
- **Resource utilization:** Review how the proposed solution will impact the resource utilization and sizing of the system.
- **Cross-feature impact:** Plan for the impact of the proposed solution on other features. For example, when customizing Accounts Receivable (A/R), consider how General Ledger (G/L) is affected. When creating your product offerings, consider how charge offers are purchased and how they are managed after being purchased.

About the BRM Client Access Libraries

BRM client applications communicate with Connection Manager (CM) by using any of the following client libraries:

- **PCM C API**. Use this API (Application Programming Interface) to build any C client application.

  See "Understanding the PCM API" and "Adding New Client Applications".
- **PCM C++ API**. Use this API to build any C++ client application.

  See "Creating Client Applications by Using PCM C++".
- **PIN library**. Use this library to manipulate data in any C or C++ applications. This library includes functions for creating and manipulating input data for the PCM API calls and functions for manipulating strings and decimal data.

  See "PIN Libraries" in *BRM Developer's Reference*.

- **Java PCM API**. Use this API to write client applications in Java that communicate with BRM.

  See "Creating Client Applications by Using Java PCM".

- **Customer Center SDK**. Use this SDK (Software Development Kit) to customize Self-Care Manager.

  See "Customizing the Self-Care Manager Interface".

- **PCMIF**. Use this Perl extension to the PCM (Portal Communication Module) library in your Perl scripts to interact with the BRM database.

  For the documentation, see "Perl Extensions to the PCM Libraries" in *BRM Developer's Reference*.

All of the client libraries are included in the BRM SDK. See "About BRM SDK" for more information.

Table 1-1 shows the APIs you can use to customize the BRM components.

Table 1-1 APIs Used to Customize BRM Components

BRM Component	PCM C	PCM C++	Java PCM	Customer Center SDK	Perl
Client applications	Yes	Yes	Yes	N/A	Yes
Self-Care Manager	N/A	N/A	N/A	Yes	N/A
BRM client tools	Yes	N/A	N/A	N/A	N/A
Policy Facility Modules	Yes	N/A	N/A	N/A	N/A
Facility Modules	Yes	N/A	N/A	N/A	N/A
Data Manager	Yes	N/A	N/A	N/A	N/A

BRM Programming Tools

You can use the following tools and utilities for customizing BRM:

- BRM SDK, which contains the APIs and libraries you need to write BRM applications, to write and customize BRM components such as FMs and DMs, and to customize BRM applications. It also includes sample applications and sample code that you can use as examples for your own work.
- Developer Center, which includes developer applications that you use to create and modify storable classes and fields, test opcodes, view objects in the database, view opcode specifications, test opcodes, and create templates that control how event data is imported into the BRM database.

2

Implementation Defaults

Learn about the default implementation of Oracle Communications Billing and Revenue Management (BRM) for business policies, such as those used for account creation and billing.

> **Caution:**
>
> Do not use or modify this product except as explicitly instructed in this documentation. Assumptions should not be made about functionality that is not documented or use of functionality in a manner that is not documented. Use or modification of this software product in any manner or for any purpose other than as expressly set forth in this documentation may result in voidance or forfeiture of your warranties and support services rights. Please consult your software license agreement for more details. If you have any questions regarding an intended use or modification of this product, please contact your BRM account executive.

Topics in this document:

- Defaults for Offering Packages and Bundles to Customers
- Defaults for Creating Customer Accounts
- Defaults for Login Names and Passwords
- Defaults for Validating Customer Account Creation Information
- Defaults for Validating Payment Information
- Defaults for Displaying and Sending Introductory Messages
- Defaults for Billing
- Defaults for Tax Calculation
- Defaults for Payments and A/R
- Defaults for Maintaining an Audit Trail of BRM Activity

Defaults for Offering Packages and Bundles to Customers

Table 2-1 lists the default settings for packages and bundles provided by BRM:

Table 2-1 BRM Default Settings

Business Decision	Default Behavior	How to Customize
Control which packages a customer can create an account with on the Web.	Customers can create accounts with packages in the default package list.	When implementing Web-based account creation, write a script that presents packages based on user input. See "Getting Data about Bundles, Charge Offers, Discount Offers, and Services" in *BRM Opcode Guide*. To customize the source code, see PCM_OP_CUST_POL_GET_PLANS.
Control which bundles are available to customers.	The customer can purchase all bundles.	When implementing Web-based account creation, write a script that presents bundles based on user input. See "Getting Data about Bundles, Charge Offers, Discount Offers, and Services" in *BRM Opcode Guide*. To customize the source code, see PCM_OP_CUST_POL_GET_DEALS.
Change the amount of time during which you can cancel a charge offer without charging a cancel fee.	Cancel fees are applied.	Change the **CancelTolerance** business parameter. See "Canceling Charge Offers without Charging a Cancel Fee" in *BRM Managing Customers*.
Set a credit limit in a package.	Credit limits are set to 0.	Change the credit limit in PDC. See "Applying Credit Limits to Balances" in *BRM Creating Product Offerings*. To customize the source code, see PCM_OP_CUST_POL_PREP_LIMIT.

Defaults for Creating Customer Accounts

Table 2-2 lists the default settings for customer accounts:

Table 2-2 Default Settings for Customer Accounts

Business Decision	Default Behavior	How to Customize
Specify the default country, in case the customer doesn't enter a country name.	USA.	Change the **country** entry in the CM configuration file. To customize the source code, see PCM_OP_CUST_POL_PREP_NAMEINFO.
Specify how to assign the merchant to each account. See "Setting Up Merchant Accounts" in *BRM Configuring and Collecting Payments*.	Use the merchant name from the **/config/ach** object.	To use multiple merchant names, customize the PCM_OP_CUST_POL_PREP_BILLINFO source code.
Specify the numbering scheme for account numbers.	Account numbers are created by combining the database number with the object ID from the account POID, for example, 0.0.0.1-1482.	To change how account numbers are generated, customize the PCM_OP_CUST_POL_PREP_ACCTINFO source code.

Table 2-2 (Cont.) Default Settings for Customer Accounts

Business Decision	Default Behavior	How to Customize
Standardize data for account names. For example, to ensure consistent display and formatting of account names, you can capitalize the first and last name, regardless of how the customer entered the information.	The name is formatted exactly how the customer entered it.	Customize the PCM_OP_CUST_POL_PREP_NAMEINFO source code.

Defaults for Login Names and Passwords

Table 2-3 lists the default settings for login names and passwords.

Table 2-3 Default Settings for Login Names and Passwords

Business Decision	Default Behavior	How to Customize
Specify the characteristics of a valid login name and password.	At least 1 character but no more than 255 characters.	To specify login requirements, including a list of "naughty words", use the Field Validation Editor. To edit the source code, see PCM_OP_CUST_POL_VALID_LOGIN and PCM_OP_CUST_POL_VALID_PASSWD.
Specify how to encrypt passwords.	Passwords for broadband services are not encrypted. AES encryption is used for nonservice passwords.	Customize the PCM_OP_CUST_POL_ENCRYPT_PASSWD source code.
Specify requirements for email login names.	Customers must use all lowercase characters for email logins. The login name must include the domain name.	Customize the PCM_OP_CUST_POL_PREP_LOGIN source code.
Specify whether the customer or the system picks the password.	Customer must specify a password.	To supply an algorithm for generating passwords, customize the PCM_OP_CUST_POL_PREP_PASSWD source code.
Specify the number of days before a CSR password must be changed.	CSR passwords expire after 90 days.	Edit the **passwd_age** entry in the CM configuration file. See "Setting the Default Password Expiry Duration" in *BRM System Administrator's Guide*.

Defaults for Validating Customer Account Creation Information

Table 2-4 lists the default settings for validating customer account creation information:

Table 2-4 Default Settings for Customer Account Creation Validation

Business Decision	Default Behavior	How to Customize
Specify the minimal information a CSR or customer must enter to create an account.	Customer must enter: • Last name • Address • City • State • ZIP code • Country • Service logins and passwords	To change which entries are required, use the Field Validation Editor. To customize the source code, see PCM_OP_CUST_POL_VALID_NAMEINFO.
Specify the formats a CSR or customer must use when entering state, phone number, and postal code.	• State: two uppercase letters, for example, CA. • Phone number: either a TAPI-compliant number or (xxx) xxx-xxxx • Postal code: (USA only); 5 digits, or 5 digits followed by a hyphen and 4 digits.	To change the required format, use the Field Validation Editor. To customize the source code, see PCM_OP_CUST_POL_VALID_NAMEINFO.
Validate state and ZIP code for tax calculation.	Do not validate state and ZIP code.	Edit the **tax_valid** entry in the CM configuration file. See *BRM Calculating Taxes*. To customize the source code, see PCM_OP_CUST_POL_VALID_NAMEINFO.

Defaults for Validating Payment Information

Table 2-5 lists for validating payment information:

Table 2-5 Defaults to Validate Payment Information

Business Decision	Default Behavior	How to Customize
Specify what information to validate when creating an account.	BRM validates the credit card only. An error in the street address is allowed. The following errors are not allowed: • Wrong postal code • Bad credit card number • Failed credit check	Customize the PCM_OP_PYMT_POL_VALIDATE source code.

Table 2-5 (Cont.) Defaults to Validate Payment Information

Business Decision	Default Behavior	How to Customize
Specify how to validate payment information, such as credit card type, expiration date, and debit account information.	For credit cards, verifies the type as: • Amex • Carte Blanche • Diners • Discover • JBC • Mastercard • Optima • VISA Validates that the credit card number has the required number of digits, and verifies the expiration date. For direct debit, validates the number of digits in bank, branch, and account numbers.	Customize the PCM_OP_CUST_POL_VALID_PAYINFO source code.
Specify whether to validate the customer's billing information during account creation.	Validates credit card information.	Change the **cc_validate** entry in the CM configuration file. See "Specifying How to Validate Customer Contact Information " in *BRM Managing Customers*. To customize the source code, see PCM_OP_PYMT_POL_SPEC_VALIDATE.
Specify whether to validate direct debit information during account creation.	Validates direct debit information.	Change the **dd_validate** entry in the CM configuration file. See "Enabling Paymentech Direct Debit Processing" in *BRM Configuring and Collecting Payments*.
Specify the time between validations of the same credit card. For example, you can allow two members of the same family to create an account for the same service without validating the credit card the second time.	Does not validate a credit card that has been validated within the last hour.	Edit the **CCRevalidationInterval** business parameter. See "Avoiding Credit Card Revalidation" in *BRM Managing Customers*. To customize the source code, see PCM_OP_PYMT_POL_SPEC_VALIDATE.
Specify the valid format for entering a credit card or debit number.	Strips spaces between groups of digits.	Customize the PCM_OP_CUST_POL_PREP_PAYINFO source code. See "Customizing Payment Method Data Preparation" in *BRM Opcode Guide*.
Specify the valid date format for credit card or debit card expiration.	BRM stores expiration dates as 4-character strings in the format *MMYY*. BRM is Y2K compliant, for example, 02 is the same as 2002. Valid formats are: • *MMYY* • *MM/YY* • *M/YY* • *MM/YYYY* All formats are converted to *MMYY*.	Customize the PCM_OP_CUST_POL_PREP_PAYINFO source code. See "Customizing Payment Method Data Preparation" in *BRM Opcode Guide*.

Table 2-5 (Cont.) Defaults to Validate Payment Information

Business Decision	Default Behavior	How to Customize
Run a checksum validation on the customer's credit card during validation.	Use a checksum to validate the credit card during account creation.	Change the **CCchecksum** business parameter. See "Disabling the Credit Card Checksum" in *BRM Opcode Guide*.

Defaults for Displaying and Sending Introductory Messages

Table 2-6 lists the default settings used to display and send introductory messages:

Table 2-6 Default Settings for Introductory Messages

Business Decision	Default Behavior	How to Customize
Specify whether to display an HTML introductory message to customers during account creation.	The introductory message is disabled by default.	Customize the default introductory message and configure your Web-based account creation to display it. See "Customizing the Introductory Message" in *BRM Opcode Guide*. To enable the introductory message, customize the PCM_OP_CUST_POL_GET_INTRO_MSG source code.
Specify whether to send a welcome email message to customers after they create an account.	The welcome email message is enabled by default.	Customize the welcome message. See "Setting up Welcome Messages to Customers" in *BRM Managing Customers*. To customize the source code, see PCM_OP_CUST_POL_POST_COMMIT.

Defaults for Billing

Table 2-7 lists the default settings for billing.

Table 2-7 Billing Defaults

Business Decision	Default Behavior	How to Customize
Set the default accounting type to open item accounting or balance forward accounting.	Balance forward accounting.	Change the **ActgType** business parameter. See "Setting the Default Accounting Type" in *BRM Configuring and Running Billing*. To customize the source code, see PCM_OP_CUST_POL_PREP_BILLINFO.

Table 2-7 (Cont.) Billing Defaults

Business Decision	Default Behavior	How to Customize
For open item accounting, set whether to include or exclude the previous total (PIN_FLD_PREVIOUS_TOTAL) of the bill from the pending amount due (PENDING_RECV) of the current bill unit (**/billinfo** object).	The previous total is not included.	Change the **open_item_actg_include_prev_total** entry in the CM.
Set the default accounting day to a specific day of the month, or to the day that the account was created.	The day that the account was created. For example, if a customer creates an account on the 15th, the accounting for that customer is done on the 15th. By default, if the account was created on day 29 - 31, the accounting day is the 1st of the next month. You can change this to be able to use any day of the month.	To set the default accounting day of month, change the **ActgDom** business parameter. See "Setting the Default Accounting Day of Month" in *BRM Configuring and Running Billing*. To customize the source code, see PCM_OP_CUST_POL_PREP_BILLINFO. To use 31-day billing, see "About Using 31-Day Billing" in BRM Configuring and Running Billing.
Set the default number of accounting cycles in a billing cycle. The billing cycle must be a whole-number multiple of accounting cycles. See "About Accounting and Billing Cycles" in *BRM Concepts*.	Monthly billing (one accounting cycle per billing cycle).	Change the **BillWhen** business parameter. See "Setting the Default Billing-Cycle Length" in *BRM Configuring and Running Billing*. To edit the source code, see PCM_OP_CUST_POL_PREP_BILLINFO.
Specify the system currency.	US dollars.	You must set the system currency when you install BRM. See "Setting the System Currency" in *BRM Managing Customers*.
Specify the default account currency for new accounts.	US dollars.	Change the **currency** entry in the CM. See "Setting the Default Account Currency" in *BRM Managing Customers*. To edit the source code, see PCM_OP_CUST_POL_PREP_BILLINFO.
Specify whether to create a long cycle or a short cycle when creating an account or changing the billing date. See "Specifying How to Handle Partial Accounting Cycles" in *BRM Configuring and Running Billing*.	If the short accounting cycle is 15 days or greater, create a short cycle. If the short accounting cycle is less than 15 days, create a long cycle.	Customize the PCM_OP_CUST_POL_PREP_BILLINFO source code.
Specify a numbering scheme for bills.	Creates the bill number B-*sequence number*, for example, "B-81".	Customize the PCM_OP_BILL_POL_SPEC_BILLNO source code.
Change the cutoff time for billing, accounting, and promotion start times. For example, if your cutoff time is 10:00:00 a.m., any events that occur after 10:00:00 a.m. on that date are included in the next billing run.	The cutoff time is midnight.	Change the **BillingCycleOffset** business parameter. See "Configuring the Billing Cutoff Time" in *BRM Configuring and Running Billing*.

Table 2-7 (Cont.) Billing Defaults

Business Decision	Default Behavior	How to Customize
Bill sponsor group member accounts.	Sponsor group member accounts are not billed.	Change the **BillingFlowSponsorship** and **BillingFlowDiscount** business parameters. See "Setting Up Billing for Charge and Discount Sharing Groups" in *BRM Configuring and Running Billing*.
Change the number of days to delay billing.	Billing is not delayed.	Change the **ConfigBillingDelay** business parameter. See "Configuring Delayed Billing" in *BRM Configuring and Running Billing*.
Change how proration is calculated: • The number of days in the cycle. • The number of days in the month.	Number of days in the cycle.	Change the **use_number_of_days_in_month** entry in the CM configuration file. See "Calculating Prorated Cycle Fees" in *BRM Configuring and Running Billing*.
Apply cycle fees in parallel for multiple services in an account.	Cycle fees is applied sequentially for each of the services in an account.	Change the **StagedBillingFeeProcessing** business parameter. See "Applying Cycle Forward Fees in Parallel" in *BRM PDC Creating Product Offerings*.
Align the purchase, cycle, and usage start and end times with the accounting cycle if you set up charge offers with delayed fees.	Start and end times are not aligned.	Change the **CycleDelayAlign** business parameter. See "Aligning Account and Cycle Start and End Times" in *BRM Configuring and Running Billing*.

Defaults for Tax Calculation

Table 2-8 lists the default settings for tax calculations:

Table 2-8 Tax Calculation Settings

Business Decision	Default Behavior	How to Customize
Enable or disable tax calculation.	Both real-time and deferred taxation are enabled.	Change the **taxation_switch** entry in the CM configuration file. See *BRM Calculating Taxes*.
Specify whether deferred taxes are calculated separately for a paying parent bill unit and its nonpaying child bill units or consolidated into a single tax item for both the parent and child bill units.	Taxes are calculated separately.	Change the **CycleTaxInterval** business parameter. See *BRM Calculating Taxes*.
Change the default ship-from locale for tax calculation.	No default: must be set, or disabled.	Change the **provider_loc** entry in the CM configuration file. See *BRM Calculating Taxes*.

Table 2-8 (Cont.) Tax Calculation Settings

Business Decision	Default Behavior	How to Customize
Defer tax calculation for all adjustments that occur at account level until the end of the billing cycle.	Tax calculation is deferred.	Change the **tax_now** entry in the CM configuration file. See "Configuring the Default Tax Method for Account Adjustments" in *BRM Calculating Taxes*.
Perform tax reversals for adjustments, disputes, and settlements that occur at the bill and account level.	Taxes are not reversed.	Change the **tax_reversal_with_tax** entry in the CM configuration file. This entry is used if the opcode does not explicitly specify the tax behavior. See "Configuring the Default Tax Method for Account Adjustments" in *BRM Calculating Taxes*.
Report zero tax amounts.	Zero tax amounts are not reported.	Change the **include_zero_tax** entry in the CM configuration file. See *BRM Calculating Taxes*.
Summarize or itemize taxes by jurisdiction.	Summarize taxes by jurisdiction.	Change the **tax_return_juris** entry in the CM configuration file. See *BRM Calculating Taxes*.

Defaults for Payments and A/R

Table 2-9 lists the default settings for payments and A/R.

Table 2-9 Payment and A/R Defaults

Business Decision	Default Behavior	How to Customize
Specify the minimum due amount that the **pin_collect** utility searches for when collecting online payments.	2.00 (expressed in the account currency).	Change the **minimum** entry in the configuration file (*BRM_home***/apps/ pin_billd/pin.conf**). See "Specifying the Minimum Payment to Collect" in *BRM Configuring and Running Billing*.
Specify the minimum due amount that custom billing utilities search for.	2.00 (expressed in the account currency).	Change the **MinimumPayment** business parameter.
Specify the minimum credit card charge.	2.00 (expressed in the account currency).	Edit the PCM_OP_PYMT_POL_PRE_COLLECT source code.
Specify the minimum refund amount.	2.00 (expressed in the account currency).	Change the **MinimumRefund** business parameter.
Use CVV fraud protection for Paymentech transactions. See "Requiring Additional Protection against Credit Card Fraud" in *BRM Configuring and Collecting Payments*.	Disabled.	Change the **Cvv2Required** business parameter.

Table 2-9 (Cont.) Payment and A/R Defaults

Business Decision	Default Behavior	How to Customize
Use card identification data (CID), a method of fraud prevention for American Express card transactions. See "Requiring Additional Protection against Credit Card Fraud" in *BRM Configuring and Collecting Payments*.	Disabled.	Change the **CidRequired** business parameter.
Specify whether to collect cycle forward and purchase fees when the customer creates an account.	Collects cycle-forward and purchase fees on account creation only for credit card customers.	Change the **CCcollect** business parameter. See "Charging Customers at Account Creation" in *BRM Managing Customers*. To customize the source code, see PCM_OP_PYMT_POL_SPEC_COLLECT.
Specify the payment methods your customers can use. If you use a payment method that is not included in the defaults, you must create a new payment method.	Accepts these payment methods: • Credit card • Invoice • Nonpaying child • Undefined • Prepaid • Debit card • Direct debit • Smart card • Beta	To create a new payment method, customize the PCM_OP_CUST_POL_PREP_PAYINFO source code. To validate payment methods, use the Field Validation Editor.
Set the payment due date for invoice payments.	30 days.	Customize the PCM_OP_CUST_POL_PREP_PAYINFO source code.
Set the default invoice type to summary invoice.	Detailed invoices are generated.	Change the value of the PIN_FLD_INV_TYPE field in the ***/payinfo*** object to **1**. • When creating a customer account, pass it in the input flist of PCM_OP_CUST_COMMIT_CUSTOMER. • When adding or changing a payment method, pass it in the input flist of PCM_OP_CUST_SET_PAYINFO.
Specify how to handle underpayments. See "Processing Overpayments and Underpayments" in *BRM Configuring and Collecting Payments*.	For balance forward accounting, BRM applies the payment to the oldest items first. If the remainder doesn't match the amount due for any one item, BRM requires that the remainder must be allocated manually. For open item accounting, BRM requires that the payment must be allocated manually.	Customize the PCM_OP_PYMT_POL_UNDER_PAYMENT source code.

Table 2-9 (Cont.) Payment and A/R Defaults

Business Decision	Default Behavior	How to Customize
Specify how to handle overpayments. See "Processing Overpayments and Underpayments" in *BRM Configuring and Collecting Payments*.	For balance forward accounting, BRM closes all open items and applies the overpayment as a credit balance. By default, you must allocate the resulting credit balance to future open items manually. For open item accounting, BRM requires that the payment must be allocated manually.	Customize the PCM_OP_PYMT_POL_OVER_PAYMENT source code.
Specify what to do if a credit card customer doesn't pay. See "Processing Late or Missed Payments" in *BRM Configuring and Collecting Payments*.	Inactivates the account if the account has an item more than 30 days past due and a credit card transaction receives one of these failures: • Soft decline • Wrong address • Wrong ZIP code • No connection Inactivates the account immediately if a credit card transaction receives one of these failures: • Bad card • Hard decline	Customize the PCM_OP_PYMT_POL_COLLECT source code.
Change the euro conversion error tolerance.	When the conversion result is less than the amount due: Apply tolerance values only if the amount applied to the item is in secondary currency. When the conversion result is more than the amount due: Do not apply tolerance values defined for balance elements.	Change the **overdue_tolerance** and **underdue_tolerance** entries in the CM configuration file. See "Handling Euro Conversion Rounding Errors" in *BRM Managing Customers*.
Collect the current direct debit balance of each account during account creation.	Collect the balance.	Change the **DDcollect** business parameter. See "Enabling Paymentech Direct Debit Processing" in *BRM Configuring and Collecting Payments*.
Specify how to handle write-off reversals.	Apply write-off reversals at the account level.	Edit the **AutoWriteOffReversal** business parameter. See "Enabling Automatic Write-Off Reversals during Payment Collection" in *BRM Managing Accounts Receivable*. You can also modify the PCM_OP_AR_POL_REVERSE_WRITEOFF opcode. See *BRM Opcode Guide*.

Defaults for Maintaining an Audit Trail of BRM Activity

Table 2-10 lists the defaults for maintaining an audit trail of BRM activity.

Table 2-10 Maintaining Audit Trail of BRM Activity

Business Decision	Default Behavior	How to Customize
Specify the BRM activity for which you want to keep an audit trail.	Keeps an audit trail of changes to customer credit card numbers and credit card expiration dates, and changes to BRM pricing components. See "Fields Marked for Auditing by Default".	Enable or disable BRM object fields for auditing by using BRM Storable Class Editor. See "Enabling Auditing for a Field".

3
Understanding Flists

Learn about flists (field lists), which pass data between Oracle Communications Billing and Revenue Management (BRM) processes.

Topics in this document:

- About Flists
- Opcode Input and Output Specifications
- About Creating and Using an Flist
- Flist Management Rules
- Flist Field Memory Management Guidelines
- Handling Errors
- Modifying the Maximum Length of an Opcode Flist Field

About Flists

The flist is the primary data structure used in BRM. Flists are containers that hold fields, each consisting of a data field name and value. Flists do not contain actual data but instead provide links to the data's location. The only exceptions are integers and strings, which are stored in flists.

Here is a simple flist example:

```
0 PIN_FLD_LAST_NAME     STR [0] "Smith"
0 PIN_FLD_FIRST_NAME    STR [0] "Joe"
0 PIN_FLD_COMPANY       STR [0] "XYZ Corporation"
0 PIN_FLD_CURRENCY      INT [0] 840
```

Many BRM processes interpret data in flist format. For example, the storage manager in the Data Manager (DM) translates flists to a format that the database can process and then translates the data from the database into an flist before passing it to the Connection Manager (CM).

BRM uses flists in these ways:

- Objects are passed as flists between opcodes or programs that manipulate the objects.
- Opcodes use flists to pass data between BRM applications and the database. For each opcode, an input flist is passed to PCM_OP, and the return flist is passed back from this routine.

In an flist, you can use any data type such as decimals, buffers, arrays, and substructures. Flists can contain any number of fields. You can place flists within other flists. Remember, though, that except for integers, data is not stored in the flist; it just links to where the data is located.

For a description of the BRM data types you can use in an flist, see "Understanding the BRM Data Types".

> **Note:**
>
> [**0**] after the field type represents the element ID. The numbers **0**, **1**, **2**, and so on at the beginning of each line indicate the field's nesting level, and **0** indicates the top level.

When you include a field in an flist, you must also include an abbreviation of the field's data type. Table 3-1 lists the valid BRM field types and their abbreviations.

Table 3-1 Flist Field Types

Field Type	Abbreviation
PIN_FLDT_ARRAY	ARRAY
PIN_FLDT_BINSTR	BINSTR
PIN_FLDT_BUF	BUF
PIN_FLDT_DECIMAL	DECIMAL
PIN_FLDT_ENUM	ENUM
PIN_FLDT_ERRBUF	ERR
PIN_FLDT_INT	INT
PIN_FLDT_POID	POID
PIN_FLDT_STR	STR
PIN_FLDT_SUBSTRUCT	SUBSTRUCT
PIN_FLDT_TSTAMP	TSTAMP

Opcode Input and Output Specifications

Each PCM opcode requires specific data to perform its operation. The opcodes take input and output data as field lists (flists), which are lists of field names and value pairs. For more information about flists, see "About Flists".

Each opcode requires its input flist to contain specific fields for operation. For example, to create an object, the PCM_OP_CREATE_OBJECT() opcode requires an input flist that includes all the fields that an object of that storable class requires.

The *Opcode Flist Reference* contains the input and output flist specifications for each opcode, defining the following parameters for each field in the flist:

- The mnemonic field names used by applications to reference the field
- The data type and size for the field
- The permissions, which specify if a field is mandatory (M) or optional (O)

The flist specifications use the following syntax to define each field in an flist:

```
class depth field (
   type = data_type
   perms = permission permission ...,
);
```

where:

- *class* specifies whether it is a field, array, or a substruct.
- *depth* contains an asterisk for each nesting level of the field.
- *field* specifies the name of the field.

Examples:

```
field PIN_FLD_NAME (
      type    =    PIN_FLDT_STR(255),
      perms   =    M,
);
array * PIN_FLD_INHERITED_INFO (
        type    =               PIN_FLDT_ARRAY,
        perms   =               O,
);
```

The flist specifications specify whether a field is mandatory or optional. XML versions of the opcode specifications are located in *BRM_home*/**apps/brm_integrations/opspecs**.

About Creating and Using an Flist

You create and manipulate flists with the flist manipulation macros in the Portal Information Network (PIN) library. You can add, remove, and modify fields using the flist field-handling macros. For more information, see "Flist Management Macros" and "Flist Field-Handling Macros" in *BRM Developer's Reference*.

Each opcode has an input and output flist specification. Create an flist for each opcode you call for the data you want to pass in. When you create an input or an output flist for an opcode, follow the flist specifications. See the flist specifications in the individual opcode descriptions.

Flists are dynamically allocated data structures. When a field is added to an flist, the value is either already dynamically allocated in memory or copied into dynamic memory as it is added.

> **Note:**
>
> Destroy the flist you create in your programs to reclaim the memory that the flist occupies. For details, see "Destroying Flists".

Adding Information to Flists

You add data to flists by replacing pointers to data using these flist management macros:

- PIN_FLIST_ELEM_PUT
- PIN_FLIST_FLD_PUT
- PIN_FLIST_SUBSTR_PUT

> **Note:**
>
> You can pass a compilation switch from your custom source code's build scripts, which causes pointers to objects that are PUT to flists to be set to NULL, preventing them from accidentally being destroyed (causing a double-free error) in later code.

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

You add data to flists by replacing the data itself using these flist management macros:

- PIN_FLIST_ELEM_SET
- PIN_FLIST_FLD_SET
- PIN_FLIST_SUBSTR_SET

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

Removing Data (Pointers) from an Flist

You remove a pointer to data from an flist and add it to another flist by using these flist management macros:

- PIN_FLIST_ELEM_TAKE
- PIN_FLIST_FLD_TAKE
- PIN_FLIST_SUBSTR_TAKE

> **Note:**
>
> These macros overwrite existing *pointers* to data, not the data itself. To free the memory used by the old data, destroy the memory location using PIN_FLIST_DESTROY_EX. Otherwise, a memory leak may occur.

You usually use the TAKE macros when you want to take data from an flist and change it.

> **Note:**
>
> When you use a TAKE macro to remove a pointer to data, you must free the memory when finished. If the memory is not freed, memory leaks may occur.

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

Copying Data from an Flist

You copy a pointer to data from one flist to another by using these flist management macros:

- PIN_FLIST_ELEM_GET
- PIN_FLIST_FLD_GET
- PIN_FLIST_SUBSTR_GET

You usually use the GET macros when you want to copy data but not change it.

> **Note:**
>
> You should treat any data that you GET using these macros as read-only because the original program may also need it.

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

Destroying Flists

You destroy an flist and free its memory by using these flist management macros in *BRM Developer's Reference*:

- PIN_FLIST_DESTROY
- PIN_FLIST_DESTROY_EX

Flists use dynamically allocated memory and must be destroyed to free that memory and prevent memory leaks. These macros first determine whether the flist has a NULL value. If so, they do nothing. If the flist exists, these macros destroy its entire contents, including all fields.

PIN_FLIST_DESTROY_EX sets the reference to the flist to a NULL value after it destroys the flist.

PIN_FLIST_DESTROY frees the memory for the flist field-value pairs, but does not set a reference to that flist to NULL. If another program subsequently attempts to destroy this flist (with freed memory but a valid flist pointer), unexpected behavior and core dumps can result.

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

Flist Creation Samples

BRM SDK includes flist creation samples in C, C++, Java, and Perl. Three samples are provided for each language: one to create a simple flist, another to create an flist with a nested array, and a third to create an flist with a substructure. For information about installing and using BRM SDK, see "About BRM SDK".

You can view the following sample programs in *BRM Developer's Reference*:

- About Using the PCM C Sample Programs
- About Using the PCM C++ Sample Programs
- About Using the PCM Java Sample Programs
- About Using the PCM Perl Sample Programs

These documents also include information about compiling and running the programs.

Using Compile-Time Flags to Avoid Errors in Flists

The PUT family of flist macros effectively transfers ownership of an object to the target flist. The object could be a decimal object, another flist, a POID, or so on. During this process, the source pointer may no longer contain a reliable reference to the original object that was PUT. For example, the target flist may subsequently be destroyed so that the pointer now refers to an area of memory that has been freed and contains garbage. Using that pointer after the PUT

may result in a crash or other undefined behavior. This is a common source of defects in programs, which can be resolved by the following idiomatic coding style:

```
PIN_FLIST_ELEM_PUT(target_flistp, source_elemp, PIN_FLD_PRODUCTS, elemid,
ebufp);
source_elemp = NULL;
```

This idiom helps avoid cases where **source_elemp** is used after the PUT where it may no longer be a valid pointer.

To reduce program errors and the burden on programmers to remember to set the source pointer variable to NULL, you can configure the BRM flist API to update the source pointer to NULL directly. This avoids errors where programmers forget to reset a pointer to NULL and simplifies the code, reducing verbosity.

To permit BRM to automatically set the source pointer to NULL during PUT operations in custom application code, set the compile-time flags in Table 3-2 in your custom code's Makefile.

Table 3-2 Compile-Time Flags for Each Flist Macro

Flist Macro	Compile-Time Flag
PIN_FLIST_SUBSTR_PUT	**-DASSIGN_NULL_AFTER_SUBSTR_PUT**
PIN_FLIST_ELEM_PUT	**-DASSIGN_NULL_AFTER_ELEM_PUT**
PIN_FLIST_FLD_PUT	**-DASSIGN_NULL_AFTER_FLD_PUT**

> **Note:**
>
> After enabling this feature, existing code may crash if it unsafely uses the original object pointer. However, this is not a cause for alarm. It simply means that the feature has exposed a potential code defect. To fix this, you can modify the code to remove the unsafe access, such as by deferring the PUT until later or reorganizing the logic to be safe with appropriate checks for a NULL pointer.

The following shows a sample error you could encounter after enabling one of the compile-time flags in your new or existing custom code and attempting to compile:

```
In file included from fm_ar_event_utils.c:31:0:
fm_ar_event_utils.c: In function 'fm_bill_adjust_event_find_adjustments':
/scratch/temp/portalbase/publish_linux_vob/publish_linux/include/pcm.h:2211:42: error:
value required as left operand of assignment
 #define SET_NULL_TO_FLD_VALUE(valp) valp = NULL;
/scratch/temp/portalbase/publish_linux_vob/publish_linux/include/pcm.h:2228:9: note: in
expansion of macro 'SET_NULL_TO_FLD_VALUE'
         SET_NULL_TO_FLD_VALUE(valp);
fm_ar_event_utils.c:1400:4: note: in expansion of macro 'PIN_FLIST_FLD_PUT'
    PIN_FLIST_FLD_PUT(read_flistp, PIN_FLD_POID, (void *)rerate_obj, ebufp);
```

Table 3-3 shows how to fix possible compilation error types in sample code using the PIN_FLIST_FLD_PUT macro.

Table 3-3 Fixing Sample Code with Compile Errors

Compilation Error Type	Problematic Code	Fixed Code
Type casting	`PIN_FLIST_FLD_PUT(srch_args_flistp, PIN_FLD_POID, (void *)s_pdp, ebufp);`	`PIN_FLIST_FLD_PUT(srch_args_flistp, PIN_FLD_POID, s_pdp, ebufp);`
Putting an object that results from a function call	`PIN_FLIST_FLD_PUT(flistp, PIN_FLD_PERCENT, (void *)pbo_decimal_round(pct, precision, ROUND_HALF_UP, ebufp);`	`pin_decimal_t *tmp_rounded_val = pbo_decimal_round(pct, precision, ROUND_HALF_UP, ebufp); PIN_FLIST_FLD_PUT(flistp, PIN_FLD_PERCENT, tmp_rounded_val, ebufp);`
Putting stack-allocated memory on an flist	`char str_msg[50]; PIN_FLIST_FLD_PUT(out_flistp, PIN_FLD_NAME, str_msg, ebufp);`	`PIN_FLIST_FLD_SET(out_flistp, PIN_FLD_NAME, str_msg, ebufp);` **Note:** Use PIN_FLST_FLD_SET to copy the string onto the flist as heap-allocated memory since **str_msg** is allocated on the stack, not the heap, and therefore becomes invalid when exiting the current lexical scope.
Putting NULL	`PIN_FLIST_FLD_PUT(flistp, PIN_FLD_STATUS, NULL, ebufp);`	`PIN_FLIST_FLD_SET(flistp, PIN_FLD_STATUS, NULL, ebufp);` **Note:** Use PIN_FLIST_FLD_SET, since NULL is not dynamically allocated memory.

Flist Management Rules

Follow these rules when creating programs that manipulate flists:

- The calling applications are responsible for allocating memory for *input* flists. This includes cases where you use a wrapper opcode to call another opcode.
- The opcode being called is responsible for allocating memory for *output* flists. For more information, see "About Creating and Using an Flist".
- The calling applications are responsible for destroying both input and output flists.
- The opcode being called is responsible for destroying both input and output flists if the opcode utilizes a wrapper opcode to call another opcode. For more information, see "Destroying Flists".
- You should never destroy an input flist within an opcode, because a calling application may need it.
- These flist management macros in *BRM Developer's Reference* allocate new memory:
  - PIN_FLIST_CREATE
  - PIN_FLIST_COPY

You must explicitly free a newly created flist using the PIN_FLIST_DESTROY_EX macro unless that flist will be used for the opcode output.

Example

You use the following syntax to copy an flist:

```
*out_flistpp = PIN_FLIST_COPY(in_flistp, ebufp);
```

Flist Field Memory Management Guidelines

You use the following guidelines when creating your programs to avoid memory problems:

- The following macros in *BRM Developer's Reference* allocate new memory for a passed value before putting it in an flist:
  - PIN_FLIST_CONCAT
  - PIN_FLIST_ELEM_ADD
  - PIN_POID_LIST_ADD_POID
  - PIN_FLIST_ELEM_COPY
  - PIN_FLIST_FLD_COPY
  - PIN_FLIST_SUBSTR_ADD
  - PIN_FLIST_ELEM_SET
  - PIN_FLIST_FLD_SET
  - PIN_FLIST_SUBSTR_SET

  These macros add or replace items in an flist by copying them; no memory ownership is transferred. When you use the SET macros, memory is allocated to copy the values and is owned by the flist. Do not explicitly free this memory.
- The following macros do *not* allocate new memory for a value before putting that value in an flist:
  - PIN_FLIST_FLD_PUT
  - PIN_FLIST_SUBSTR_PUT
  - PIN_FLIST_ELEM_PUT

  These macros add or replace items in the flist, storing the data previously owned by the caller. The allocation memory is transferred to the flist. These macros link the memory occupied by the value to the named flist. You *cannot* apply these for local scope (auto) variables if you want to put them into the return flist.
- The following macros take ownership of the memory owned by the flist for a retrieved value:
  - PIN_FLIST_FLD_TAKE
  - PIN_FLIST_SUBSTR_TAKE
  - PIN_FLIST_ELEM_TAKE
  - PIN_FLIST_ELEM_TAKE_NEXT
  - PIN_POID_LIST_TAKE_NEXT_POID
  - PIN_POID_LIST_REMOVE_POID

These macros remove items from the flist, return pointers, and turn ownership of the allocated memory over to the caller.

- The following macros do *not* allocate new memory for a retrieved value:
  - PIN_FLIST_ELEM_DROP
  - PIN_FLIST_FLD_DROP
  - PIN_FLIST_SUBSTR_DROP
  - PIN_FLIST_ANY_GET_NEXT
  - PIN_FLIST_FLD_GET
  - PIN_FLIST_ELEM_GET
  - PIN_FLIST_ELEM_GET_NEXT
  - PIN_FLIST_SUBSTR_GET

  A pointer to the allocated segment of memory that belongs to the flist returns. To maintain the integrity of the flist, you should *never* apply the PIN_FLIST_DESTROY macro to an flist pointer returned by these macros. The flist retains ownership of its allocated memory.
- The following macros move fields from one flist to another:
  - PIN_FLIST_FLD_MOVE
  - PIN_FLIST_ELEM_MOVE

  Memory ownership of the field changes from the source flist to the destination flist.
- The following field management macros allocate new memory:
  - pbo_decimal_abs
  - pbo_decimal_add
  - pbo_decimal_copy
  - pbo_decimal_divide
  - pbo_decimal_from_double
  - pbo_decimal_from_str
  - pbo_decimal_multiply
  - pbo_decimal_negate
  - pbo_decimal_round
  - pbo_decimal_subtract
  - pbo_decimal_to_str
  - PIN_POID_COPY
  - PIN_POID_CREATE
  - PIN_POID_DESTROY
  - PIN_POID_LIST_COPY
  - PIN_POID_LIST_COPY_NEXT_POID
  - PIN_POID_LIST_COPY_POID
  - PIN_POID_LIST_CREATE
  - PIN_POID_LIST_DESTROY

  All of the preceding macros return the memory owned to the caller.

- The memory allocated by the macros in these guidelines is owned by the flist. Typically, all flists created by code must be destroyed using PIN_FLIST_DESTROY or PIN_FLIST_DESTROY_EX.
- Any memory allocated using pin_malloc must be freed unless it is added to an flist using a PUT macro.
- All flists must be destroyed eventually, either directly or by nesting them in another flist as a child and giving memory ownership to that flist.
- In Facilities Modules (FMs), the output flist is the only flist that does not need to be explicitly destroyed by the FM, because the Connection Manager (CM) framework destroys it after use.

For details on these macros, see "Flist Management Macros" in *BRM Developer's Reference*.

Handling Errors

All flist routines take a pointer to **pin_ebuf_t** as the final argument. This pointer, called **ebufp**, must point to a preallocated **pin_ebuf_t** into which error information is written.

The BRM APIs use the **pin_ebuf_t** structure to pass back detailed information about the error. BRM includes standard logging routines that print formatted log messages using the date in **pin_ebuf_t**. You can use these log messages to determine where the error occurred.

See "Finding Errors in Your Code" for more information on how BRM handles error messages.

Modifying the Maximum Length of an Opcode Flist Field

You can change the maximum length of an opcode's flist field, such as from 128 characters to 255 characters. You do so by:

- Generating a custom opcode specification for the opcode using the **generate_config_xml_from_opcode_spec.py** script.
- Loading the custom opcode specification into the **/config/maxlength_validate** database object using the **load_config** utility.

To modify the maximum length of an opcode flist field:

1. Copy the opcode specification from *BRM_home***/apps/brm_integrations/opspecs** to *BRM_home***/sys/data/config**.
2. Change to the *BRM_home***/sys/data/config** directory and edit the opcode specification.

   For example, to increase the size of the PCM_OP_CUST_COMMIT_CUSTOMER opcode's PIN_FLD_DESCR input flist field, you would:

   a. Copy the **PCM_OP_CUST_COMMIT_CUSTOMER.xml** file from *BRM_home***/apps/brm_integrations/opspecs/** to *BRM_home***/sys/data/config** and edit it there.

   b. In the PIN_FLD_DESCR line, change the **maxlen** attribute value:

   ```
   <String name="PIN_FLD_DESCR" use="optional" maxlen="1023">
   ```

   c. Save and close the XML file.
3. Run the **generate_config_xml_from_opcode_spec.py** script:

   ```
   generate_config_xml_from_opcode_spec.py opcodeName.xml
   ```

where *opcodeName* is the opcode name. For example, if you are modifying the PCM_OP_CUSTOM_COMMIT_CUSTOMER opcode specification, you would enter the following:

```
generate_config_xml_from_opcode_spec.py PCM_OP_CUST_COMMIT_CUSTOMER.xml
```

If successful, the script generates a **config**_*opcodeAbbrev*.**xml** file where *opcodeAbbrev* is the name of the opcode without PCM_OP_. For example, it generates the **config_CUST_COMMIT_CUSTOMER.xml** file for the PCM_OP_CUST_COMMIT_CUSTOMER opcode.

See "generate_config_xml_from_opcode_spec.py" for information about the utility's syntax and parameters.

4. Load the XML file into the database by running the **load_config** utility:

```
load_config config_opcodeAbbrev.xml
```

If the XML file is in a different directory than the one from which you are running the utility, specify the full path to the file.

See "load_config" for information about the utility's syntax and parameters.

The configuration is loaded in the **/config/maxlength_validate** object.

4

Understanding Storable Classes

Learn about storable classes, which define the storable objects that store data in the Oracle Communications Billing and Revenue Management (BRM) database.

Topics in this document:

- About Storable Classes and Objects
- Reading Objects
- Creating Objects
- Writing Fields in Objects
- Incrementing Fields in Objects
- Deleting Objects
- Deleting Fields in Objects
- Managing a Large Number of Objects
- Improving Performance when Working with Objects
- Locking Specific Objects
- Disabling Granular Object Locking

About Storable Classes and Objects

Storable classes define the storable objects that store data in the BRM database. A storable class is a template consisting of a collection of fields in the BRM database. Objects are instances of storable classes.

Objects are stored persistently in SQL tables in the BRM database. To understand how objects are mapped to SQL tables, see "Storable Class-to-SQL Mapping" in *BRM Developer's Reference*.

Objects are passed in the form of flists between opcodes or programs that use the objects. For detailed information about flists, see "Understanding Flists".

You manipulate the objects by using opcodes in the Portal Communication Module Application Programming Interface (PCM API). For more information about the PCM API, see "Understanding the PCM API".

Each object has a unique Portal object ID (POID). The POID identifies the object in the BRM database.

The POID contains the following information:

database_number object_type object_id object_revision_level

Example:

```
0.0.0.1 /account 1234 0
```

Storable Class Naming and Formatting Conventions

BRM uses the following conventions for storable class names and definitions:

- A storable class name begins with a forward slash (/). For example: **/config**.
- Each slash in a storable class name represents a level of class inheritance. For example **/config/adjustment** is a subclass of **/config** and **/config/adjustment/event** is a subclass of **/config/adjustment**.

Subclassing

You can subclass a storable class to extend its functionality. When you subclass a storable class, the subclass inherits all the fields defined in the parent class and can have additional fields specific to the subclass. For example, the **/service/email** storable class contains all the information in the **/service** storable class plus additional information, such as login name and email address, specific to the email service.

Base storable classes are parent classes predefined in BRM from which you can create subclasses to add new functionality. Subclasses of storable classes can be extended or unextended:

- An *extended subclass* inherits all the fields defined in its base storable class and contains new fields that add functionality specific to the subclass, as illustrated by the **/service/email** subclass.

  You can create a subclass of an extended subclass to add functionality to an extended subclass. An example of this type of subclass is **/config/adjustment/event**.
- An *unextended subclass* inherits all the fields defined in its base storable class but does not contain any new fields for additional functionality. The only difference between an unextended subclass and its base storable class is the storable class type.

  The unextended subclasses are primarily used for tracking and for grouping subclasses with similar functionality. For example, **/event/billing** is a subclass of **/event**, but it does not contain any additional fields. It is primarily used to group different BRM billing events, such as charge, debit, and payment.

When you subclass a predefined base storable class or a subclass by adding new fields, your new storable class has the following characteristics:

- The same fields as its original base storable class with the attributes of that class.
- Any new fields you add.
- The common defined behavior and functionality of storable classes in BRM.

About Defining Storable Classes

You use fields to define storable classes. Fields in storable classes have corresponding fields in opcode input flists. The fields in the objects receive specific instructions and behavior from the corresponding fields in the opcode that manipulates the object.

Each field has the following parts:

- A mnemonic *name* that describes its function. For example, **PIN_FLD_LOGIN**. Field names are unique within an flist. Applications must use this name to refer to the field.

  The default fields used in BRM start with **PIN_FLD_**. You can use a different prefix for your custom fields to distinguish them from the default fields.

- A *type* that specifies its data type and defines the range of values it can accept. For example, **PIN_FLDT_STR** for string type.

  For information on the BRM data types, see "Understanding the BRM Data Types".

- *Permission*, which specifies if the field is optional, mandatory, writable, and so on.
- An *ID number* that establishes it in the data dictionary. When you define a new field, the field is added to the data dictionary and is assigned the next consecutive ID number in the ranges reserved for customer use. For example, if the ID of the last field you created is 10,500, your new field's ID number is 10,501.

  Table 4-1 lists the field ID ranges for Oracle-only use and customer use.

Table 4-1 BRM Field ID Restrictions

Field ID Range	Reserved For
0 through 9999	Oracle use only
10,000 through 999,999	Customer use
1,000,000 through 9,999,999	Oracle use only
Over 10,000,000	Customer use

Fields Common to All Storable Classes

Every BRM storable class requires the following fields for its object to be created in the system:

- PIN_FLD_POID, which contains the POID. See "Portal Object ID (POID)" for more information.
- PIN_FLD_NAME, which contains the name of the storable class.
- PIN_FLD_CREATED_T, which contains the date the object was created.
- PIN_FLD_MOD_T, which contains the date the object was modified.
- PIN_FLD_READ_ACCESS, which specifies the read permissions for the object.
- PIN_FLD_WRITE_ACCESS, which specifies the write permissions for the object.

These fields are available to the BRM applications and Facilities Modules (FMs), but you cannot write to them. They can be manipulated only by a Data Manager (DM).

Defining New Fields for Storable Classes

You define new fields and add them to the data dictionary by using the Storable Class Editor. After you define a field, you use the **PIN_FLD_MAKE** macro to create an encoded number for the field. You use this number to build flists for opcode and object manipulation in C code. The **PIN_FLD_MAKE** macro is defined in *BRM_home***/include/pcm.h**, where *BRM_home* is the directory in which the BRM server software is installed.

For more information on creating custom fields and storable classes, see "Creating Custom Fields".

For information on defining new fields in C++, see "Creating Custom Fields".

For information on defining new fields in Java, see "Using Custom Fields in Java Applications".

Reading Objects

There are two ways to read the contents of objects in a BRM database:

- Use Object Browser, part of Developer Center, to view objects in the BRM database. You can see a list of all the objects in a storable class and the contents of objects that you select. You can also save and print object contents.
- You can run opcodes either programmatically or in a test application such as **testnap**.

You can use opcodes to read objects in the database by using the read operations included in BRM. These opcodes are optimized for the most frequently used operations in BRM. You can read only a portion of the fields in an object instead of the complete object.

Use one of these opcodes to read objects:

- PCM_OP_READ_OBJ loads and returns the entire object, no matter how many rows and tables the object spans. See "Reading an Entire Object".
- PCM_OP_READ_FLDS loads only the fields requested. It accesses fewer tables than when an entire object is read. See "Reading Fields in an Object".

Locking Objects when Reading them

You can lock an object to avoid an extra round trip to the database. To lock an object while the PCM_OP_READ_OBJ or PCM_OP_READ_FLDS opcode reads it, use the PCM_OPFLG_LOCK_OBJ flag. The query is turned into the equivalent of a select for update, which places an exclusive lock on the rows in the database.

When using the PCM_OPFLG_LOCK_OBJ flag during an opcode process, the flag works as follows:

- If the Facilities Module (FM) opcode opens a read/write transaction, the PCM_OPFLG_LOCK_OBJ flag locks the object for a PCM_OP_READ_OBJ or PCM_OP_READ_FLDS operation.
- If the FM opcode opens a read-only transaction and then tries to use PCM_OPFLG_LOCK_OBJ in any subsequent read calls, the DM returns an error.
- If the FM opcode does not open a transaction and then tries to use PCM_OPFLG_LOCK_OBJ in any subsequent read calls, the DM returns an error.

Most opcode operations lock the account when they begin processing. Though this provides reliable data consistency, locking an account locks all of its associated objects and can prevent other opcodes from operating on them. This can decrease the throughput of the system. To alleviate this problem in affected systems, you may choose to lock the specific objects an opcode will change instead of the whole account; the objects an opcode does not change can still be accessed by other opcodes. See "Locking Specific Objects".

Reading an Entire Object

To read an entire object from the database, use the PCM_OP_READ_OBJ opcode. Specify the POID of the object to read in the input flist. The POID of the object and all fields in the object return, including array elements and substructures.

Audit trail information

PCM_OP_READ_OBJ checks for an audit flag to see whether the read request is for an audited object. If an audit flag is set in the call to this opcode, a search for the audit trail is performed.

For information about accessing audit trails, see "Accessing Audit Trail Information".

When a field is marked for auditing, a shadow object is created every time the field is modified. A shadow object is a replica of the original object that contains the POID revision number of the object before it was modified.

For more information about shadow objects, see "About Shadow Objects".

To retrieve audit-trail revisions from the database, use these flags in the call to this opcode to send a request to the database DM:

- PCM_OPFLG_USE_POID_GIVEN

  Runs a search-and-read operation for the POID you specify, which is the POID of the shadow object in the audit trail.

- PCM_OPFLG_USE_POID_NEAREST

  Runs a search-and-read operation for the audit-trail revision number that you specify (the POID of the shadow object). If the exact POID is not found, it finds the POID with the revision number immediately preceding the revision number of the POID you specify.

- PCM_OPFLG_USE_POID_NEXT

  Runs a search-and-read operation for the shadow object POID that contains the revision number next higher than the revision number in the POID you specify.

- PCM_OPFLG_USE_POID_PREV

  Runs a search-and-read operation for the shadow object POID that contains the revision number immediately preceding the revision number in the POID you specify.

Reading Fields in an Object

To read one or more fields in an object, use the PCM_OP_READ_FLDS opcode. This opcode returns the POID of the object from which the fields were read, along with the specified fields and their values.

This opcode returns the POID of the object from which the fields were read, along with the specified fields and their values.

PCM_OP_READ_FLDS allows a client application to read specified fields in an object. Specify the POID of the object along with the list of fields to be read in the input flist. The POID is mandatory; the fields are optional. If there are no fields present, only the POID is read and returned.

To read an array element, specify the element in the input flist. If the array element contains a substructure that contains fields you want to read, you must also specify the substruct elements in the input flist.

> **Note:**
>
> To improve performance, some object arrays are stored in a special serialized format. When a client application requests fields from a serialized array, PCM_OP_READ_FLDS returns the entire array rather than just the specified fields.

You can read all the fields of a substructure or array in the following ways:

- To read an entire substruct, put the substruct field in the input flist with a **NULL** value.
- To read all the fields in an array element, put the array element in the input flist with a **NULL** value. This returns all fields, including those from substructs.
- To read all the elements of an array, put the array element in the input flist with an element ID of PIN_ELEMID_ANY.

Examples

- This opcode can be used with a read-only transaction to obtain a frozen view of an object. When a read-only transaction is open, all reads are performed on fields in the state they are in when the transaction is opened. This is an important consideration when you perform multiple reads and must ensure that the data does not change between reads, such as when adding up fields for a balance.

  For information about opening a read-only transaction, see "Using Transactions in Your Client Application".

- You can use this opcode to find the value of a flag. This example shows the input and output flists for reading the value of the PIN_FLD_FLAGS field in the PIN_FLD_BALANCES array element in a **/balance_group** object:

```
CM input flist: op PCM_OP_READ_FLDS, flags 0x0
# number of field entries allocated 20, used 2
0 PIN_FLD_POID                POID [0] 0.0.0.1 /balance_group 685398 3
0 PIN_FLD_BALANCES           ARRAY [840] allocated 20, used 1
1     PIN_FLD_CREDIT_PROFILE    INT [0] 0

CM output flist: op PCM_OP_READ_FLDS
# number of field entries allocated 20, used 2
0 PIN_FLD_POID                POID [0] 0.0.0.1 /balance_group 685398 1
0 PIN_FLD_BALANCES           ARRAY [840] allocated 20, used 1
1     PIN_FLD_CREDIT_PROFILE    INT [0] 22
```

Creating Objects

To create an object, use the PCM_OP_CREATE_OBJ opcode. This opcode creates a new object of the type specified in the input flist. You must specify the database type and storable class type subfields of the object POID on the input flist. The POID ID is ignored unless you use the PCM_OPFLG_USE_POID_GIVEN flag.

The PCM_OP_CREATE_OBJ input flist must include all fields the object requires. The fields required in the input flist depend on the object type. For example, an **/account** object requires the security code, account number, account type, balances, and credit fields.

This opcode returns the POID of the object created. If you use the PCM_OPFLG_READ_RESULT flag, it also returns all fields from the created object, including array elements and substructures.

PCM_OP_CREATE_OBJ Opcode Flags

Use these flags when creating objects:

- PCM_OPFLG_USE_POID_GIVEN

  Uses the POID ID you specify in the input flist. If the ID is a duplicate, a new POID ID is assigned. Not all object POID IDs can be assigned by a user. For information on which

object POID IDs can be assigned, see the object specifications in "Storable Class Definitions" in *BRM Developer's Reference*.

When this flag is not passed, the opcode assigns a new POID ID.

- PCM_OPFLG_READ_RESULT

  Returns all fields from the created object, including array elements and substructures.

- PCM_OPFLG_CACHEABLE

  Enables caching each transaction's objects in the Connection Manager (CM) instead of writing them immediately to the database. See "Improving Performance when Working with Objects".

  If the flag is not passed, the opcode writes the input flist to the database immediately and writes it to the cache for future use.

- PCM_OPFLG_ADD_MESSAGE

  Performs enqueuing operations to the Oracle Advanced Queuing (AQ) database queues rather than writing objects to the database. See "Configuring Your AQ Database Queues" in *BRM System Administrator's Guide*.

Writing Fields in Objects

To write fields in an object, use the PCM_OP_WRITE_FLDS opcode. This opcode returns the POID of the object whose fields were written, including the new revision number. This opcode allows a client application to set the values of fields in an object. Specify the fields and values to set, along with the POID of the object, on the input flist. You must update at least one field.

The field values are absolutely set to the values you provide. To make a relative change to a numeric field value, use the PCM_OP_INC_FLDS opcode.

Update array element fields by specifying the array element ID and the field element ID in the input flist. If there is no array element for the fields you want to update, you must create the element by using the PCM_OPFLG_ADD_ENTRY flag.

> **Note:**
>
> This flag is required because BRM allocates space for array elements only when they are created and assigned a value. If you do not use the PCM_OPFLG_ADD_ENTRY flag, you receive an error when you try to update array fields in an element that doesn't exist.

The PCM_OPFLG_ADD_ENTRY flag is required only for array fields. Fields that are not part of an array element must already exist (assuming the object exists) so they are updated as requested.

If you set an array element in the input flist and use the element ID PIN_ELEMID_ASSIGN, the element is created with an element ID numbered next higher than the highest existing element ID for that array.

Not all fields in each object are writable by the application. For details on which fields are writable, see "Storable Class Definitions" in *BRM Developer's Reference*.

PCM_OP_WRITE_FLDS Opcode Flags

Use these flags:

- PCM_OPFLG_ADD_ENTRY

  Creates an array element and updates fields as requested. If the array element already exists, this flag is ignored. PCM_OPFLG_ADD_ENTRY cannot be used to create ordinary fields.

- PCM_OPFLG_READ_RESULT

  Returns all fields from the created object, including array elements and substructures.

- PCM_OPFLG_NO_RESULTS

  Although this flag can be included, the PCM_OP_WRITE_FLDS opcode ignores it and returns the POID.

- PCM_OPFLG_CACHEABLE

  Enables caching each transaction's objects in the CM instead of writing them immediately to the database. See "Improving Performance when Working with Objects".

  If the PCM_OPFLG_CACHEABLE flag is not set, the opcode writes the input flist to the database immediately and writes it to the cache for future use.

  If an array element in the input flist has an **element ID** of PIN_ELEM_ID_ASSIGN, the element is not cached, even if the PCM_OPFLG_CACHEABLE flag is set.

  If neither PCM_OPFLG_CACHEABLE nor PCM_OPFLG_ADD_ENTRY is set, and the array entry does not exist, the opcode fails. If PCM_OPFLG_CACHEABLE is set but PCM_OPFLG_ADD_ENTRY is not set, and the array entry does not exist, the opcode also fails, but it does not return the error immediately. The delayed error appears when the fields are actually written to the database.

Incrementing Fields in Objects

To increment fields in an object, use the PCM_OP_INC_FLDS opcode. This opcode returns the POID of the object whose fields were updated, including the new revision number. It also returns the revised values of the selected fields unless the PCM_OPFLG_NO_RESULTS flag is used.

This opcode increments or decrements fields specified in the input flist. Only fields of type PIN_FLDT_INT and PIN_FLDT_DECIMAL can be incremented or decremented, and both types must be signed in the input flist. The signed value of the field in the input flist determines whether the field is incremented or decremented.

You must update at least one field. Specify the POID of the object that contains the fields to update, along with at least one field, in the input flist.

Update array element fields by specifying the array element ID along with the field element ID in the input flist. If there is no array element for the fields you want to update, you must create the element by using the PCM_OPFLG_ADD_ENTRY flag.

> **Note:**
>
> This flag is required because BRM allocates space for array elements only when they are created and assigned a value. If you do not use the PCM_OPFLG_ADD_ENTRY flag, you receive an error when you try to update array fields in an element that doesn't exist. An error also occurs if the array contains any nonincremental fields or mandatory fields of a nonincremental type.

The PCM_OPFLG_ADD_ENTRY flag is required only for array fields. Fields that are not part of an array element must already exist (assuming the object exists) so they are updated as requested.

If you set an array element on the input flist and assign the element ID a value of PIN_ELEMID_ASSIGN, the element is created with an element ID that is numbered next higher than the highest existing element ID for that array.

Updating Decimal Data Types

When you increment or decrement fields of decimal data type, the result depends on the value of the field both in the database and in the input flist. If the value of the field in the database is NULL, BRM converts that value to 0 before updating. If the value of the field in the input flist is NULL or 0, no action is taken on the value in the database. This prevents a non-NULL value in the database from being converted to NULL.

Table 4-2 shows the three possible results of an increment (0, NULL, or non-NULL) for all possible field value combinations. These results apply only to decimal data types:

Table 4-2 Incrementing Database Field Values

Database Field Value	Flist Increment Value	Result of Increment
NULL	NULL or 0	NULL
NULL	Non-NULL	Non-NULL
0	NULL or 0	0
0	Non-NULL	Non-NULL
Non-NULL	NULL or 0 or non-NULL	Non-NULL

If you want a field value to remain open (such as a credit limit), you should increment the field by a value of either 0 or NULL.

You can prevent the conversion of the field value in the database from NULL to 0 by using the PCM_OPFLG_USE_NULL flag. This flag assigns a NULL value to the field in the input flist, which prevents a change to the database value.

Updating Integer Data Types

You cannot assign a **NULL** value to fields of type **INT** in the input flist. If you update an **INT** data type, the PCM_OPFLG_USE_NULL flag is ignored. To maintain a NULL value in the database for a field of type **INT**, you must increment with **0**. If the increment value is nonzero in the input flist, the result is always nonzero.

Not all fields in each object can be incremented by an application. For information on which fields can be incremented, see "Storable Class Definitions" in *BRM Developer's Reference*.

PCM_OP_INC_FLDS Opcode Flags

Use these flags:

- PCM_OPFLG_ADD_ENTRY

  Creates an array element, if it doesn't already exist, and updates the specified fields as requested. It cannot be used to create nonarray fields. If the array element already exists, this flag is ignored.

- PCM_OPFLG_READ_RESULT

  Returns all fields from the created object, including array elements and substructures.

- PCM_OPFLG_USE_NULL

  Prevents a NULL field value in the database from being converted to 0 and updated.

- PCM_OPFLG_NO_RESULTS

  Returns only the POID, without the updated data. This flag provides higher performance by skipping the extra processing of returning updated data.

Deleting Objects

To delete an object, use the PCM_OP_DELETE_OBJ opcode. Specify the POID of the object to delete in the input flist. This opcode ignores the field values of the object to be deleted. When an object is deleted, its POID ID cannot be reused.

This opcode deletes only the object passed in. No integrity checks are performed to ensure that the object is not referenced in any way.

> **Note:**
>
> To maintain database consistency, you must make sure that your application deletes any other objects that reference or are referenced by the object you delete.

This opcode returns the POID of the deleted object.

Deleting Fields in Objects

To delete fields in an object, use the PCM_OP_DELETE_FLDS opcode. This opcode returns the POID of the object from which an element was deleted, including the new revision number.

You must delete at least one array element. Specify the POID of the object from which to delete elements in the input flist. Also specify the array element ID for each element to be deleted. To delete an entire array, put the array in the input flist and use the element ID PCM_RECID_ALL.

The value for each array element to be deleted must be NULL. The value of fields within an array element to be deleted are ignored. You cannot delete fields within an array element without deleting the array element.

Only optional arrays and array elements can be deleted. Attempting to delete a mandatory element will return an error. You cannot delete fields that are not part of an array element, and you cannot delete arrays or array elements within a substructure.

Managing a Large Number of Objects

In addition to working with individual objects, you can use base opcodes to perform the following operations on a large number of objects in one database operation instead of accessing the database for every object:

- Create a large number of objects of the same type. See "Creating a Large Number of Objects".
- Update fields in a large number of objects. See "Editing a Large Number of Objects".
- Delete a large number of objects of the same type. See "Deleting a Large Number of Objects".

Creating a Large Number of Objects

To create a large number of objects of the same type, use the PCM_OP_BULK_CREATE_OBJ opcode.

The PCM_OP_BULK_CREATE_OBJ input flist must include all the fields that the objects of the storable class requires. The required input flist fields depend on the type of objects that you are creating; for example, an **/account** object includes security code, account number, account type, balances, and credit fields. You can use the PIN_FLD_COMMON_VALUES array in the input flist for fields that have common values for all objects being created.

> **Note:**
>
> You cannot use this opcode to create a new storable class. You can use it only to create objects in an existing storable class.

You can use the same flags with this opcode as you can with the PCM_OP_CREATE_OBJ opcode, except that the following flags are not supported:

- PCM_OPFLG_USE_POID_GIVEN
- PCM_OPFLG_READ_RESULT

This opcode returns the type-only POID of the objects created, but it does not return the full POIDs of the individual objects created.

Editing a Large Number of Objects

To update fields in a large number of objects of the same type, use the PCM_OP_BULK_WRITE_FLDS opcode.

This opcode updates the values of the fields or adds new fields in the objects that meet the conditions you specify in the input flist. It returns the POID type and count of the objects.

To update array element fields, specify the array element ID and the field element ID in the input flist. You can update first- and second-level array fields. If there is no array element for

the fields you want to update, you must create the element by using the PCM_OPFLG_ADD_ENTRY flag.

> **Note:**
>
> This flag is required only for array elements because BRM allocates space for array elements only when they are created and assigned a value. If you do not use the PCM_OPFLG_ADD_ENTRY flag, you receive an error when you try to update array fields in an element that doesn't exist.
>
> Fields that are not part of an array element must already exist for an object, and they are updated.

If you set an array element in the input flist and use the element ID PIN_ELEMID_ASSIGN, the element is created with an element ID higher than the highest existing element ID for that array.

Deleting a Large Number of Objects

To delete a large number of objects of the same type, use the PCM_OP_BULK_DELETE_OBJ opcode. This opcode deletes the objects of the type that meet the conditions you specify in the **where** clause of the query. It returns the POID type and the count of the objects deleted.

> **Note:**
>
> - You cannot use the opcode to delete specific fields in an object, only to delete complete objects.
> - You cannot use the opcode to delete an entire storable class, only to delete objects in that class.

Locking Objects when Editing or Deleting a Large Number of Objects

To enforce the data integrity of bulk operations (the PCM_OP_BULK_WRITE_FLDS and PCM_OP_BULK_DELETE_OBJ opcodes), specify the PCM_OPFLG_LOCK_OBJ flag for the opcodes.

> **Note:**
>
> The PCM_OP_LOCK_DEFAULT flag is ignored by the bulk opcodes.

When the PCM_OPFLG_LOCK_OBJ flag is specified, the balance groups of the known objects or the unknown objects will be locked. The rules for identifying which balance groups are locked are identical to the rules used in "Locking Specific Objects".

Improving Performance when Working with Objects

Some opcodes used for searching and manipulating objects use the PCM_OPFLG_CACHEABLE flag. This flag enables caching each transaction's objects in the Connection Manager (CM) instead of writing them immediately to the database. Caching makes the CM and the DM more efficient because the CM doesn't request the DM to write the same object to the database multiple times.

Objects are cached at the end of a transaction unless they must be written to the database earlier. For example, a search causes immediate execution of all pending writes so that the search can work on the most current data.

If the PCM_OPFLG_CACHEABLE flag is not set, the opcode immediately writes the input flist to the database and then to the cache for future use.

The CM writes flist fields to the database when the application does the following:

- Runs one of the following opcodes:
  - PCM_OP_CREATE_OBJ
  - PCM_OP_SEARCH
  - PCM_OP_STEP_SEARCH
  - PCM_OP_STEP_NEXT
  - PCM_OP_GLOBAL_SEARCH
  - PCM_OP_GLOBAL_STEP_SEARCH
  - PCM_OP_GLOBAL_STEP_NEXT
  - PCM_OP_READ_OBJ
  - PCM_OP_READ_FLDS
  - PCM_OP_WRITE_FLDS
- Runs PCM_OP_INC_FLDS, and the fields are part of the object in the writable cache.

Sometimes, the object can be partially available in the cache due to previous executions of PCM_OP_READ_FLDS or PCM_OP_WRITE_FLDS. If the partial object is in the read-only cache, the CM destroys it and reads a complete object from the database through the DM. The CM caches that object before returning it to the application. If the partial object has been updated and therefore is in the writable cache, the CM writes it to the database before reading and caching the complete object.

The scope of a transaction cache is one transaction.

> **Note:**
>
> Transaction caching is not always beneficial. For example, if a transaction reads a given object only once, it should not use the PCM_OPFLG_CACHEABLE flag.

Locking Specific Objects

Locking on an account level (higher object hierarchy) can cause contention between opcodes and therefore create a bottleneck, reducing the throughput of the BRM system. Because the

default locking process locks at the account level, performance can be improved by locking only the working objects rather than the account.

You can lock balance groups for the specific operations that are impacting performance. If you do not specify a more granular locking procedure, transactions lock account objects; no code change is required.

There are some frequently used operations that can take advantage of granular locking on balance group objects. They may lock the object's default or associated balance group to improve performance.

> **Note:**
>
> Exercise caution when customizing locking strategies. Changes to lower-level locking on balance group objects may cause a deadlock if you are not familiar with the execution paths of the Facilities Module (FM) utility subroutines.

BRM supports balance group locking for transactions and opcodes that open the following known objects:

- A given account
- A given service
- A given group
- A given bill unit (**/billinfo** object)
- A given bill
- A given balance group
- A given profile
- A purchased charge offer
- A purchased discount offer
- An event
- A journal

You use the following flags to lock balance groups:

- Use the PCM_TRANS_OPEN_LOCK_OBJ flag to lock all the associated balance groups (multiple balance groups) during the opening of a transaction. The equivalent flag for use with the other base opcodes is PCM_OPFLG_LOCK_OBJ.

  For example:

  ```
  fm_utils_trans_open(ctxp, opflags|PCM_TRANS_OPEN_LOCK_OBJ, pdp, ebufp);
  ```

- Use the PCM_TRANS_OPEN_LOCK_DEFAULT flag to lock the default balance group during the opening of a transaction. The equivalent flag for use with the base opcode is PCM_OPFLG_LOCK_DEFAULT.

  For example:

  ```
  PCM_OP(ctxp, PCM_OP_READ_FLDS, PCM_OPFLG_LOCK_OBJ, s_flistp, &o_flistp, ebufp);
  ```

Using these flags can improve your system's performance if many balance groups are associated with the object you are targeting with the lock request. Each flag locks balance groups according to the object type opened by the opcode or utility as shown in Table 4-3:

Table 4-3 Balance Locking Flags

Object Types	PCM_TRANS_OPEN_LOCK_OBJ	PCM_TRANS_OPEN_LOCK_DEFAULT
Account	Generate a SQL template to do bulk locking on all the account's balance groups.	Lock the account's default balance group.
BillInfo	Generate a SQL template to do bulk locking on all the bill unit's balance groups.	Lock the bill unit's default balance group.
Balance Group	Lock the balance group. **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the balance group.
Service	Lock the service's default balance group. **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the service's default balance group.
Profile	Lock all the parent account's balance groups (alternatively). **Note:** There may be some ambiguity in this rule.	Lock the parent's default balance group.
Group	Lock all the owner account's balance groups.	Lock the owner account's default balance group.
Bill	Lock all bill unit's balance groups.	Lock the bill unit's default balance group.
Item	Lock the item's default balance group. **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the item's default balance group.
Journal	Lock the journal object as is by returning its POID. **Note:** This rule is identical to that of the PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the journal object as is by returning its POID.
Event	Lock the event object as is by returning its POID. **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the event object as is by returning its POID.
Purchased Charge Offer	Lock all of the service's default balance groups (if any) or parent account's default balance groups (alternatively). **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock all the service's default balance groups (if any) or parent account's default balance groups (alternatively).
Purchased Discount Offer	Lock all of the service's default balance groups (if any) or parent account's default balance groups (alternatively). **Note:** This rule is identical to that of PCM_TRANS_OPEN_LOCK_DEFAULT.	Lock the all the service's default balance groups (if any) or parent account's default balance groups (alternatively).
Others	Lock all the balance groups of the parent account, provided the account is not the root or the PCM login account. Otherwise, lock the object as is. **Note:** BRM assumes that the login user ID is an account POID.	Lock the default balance group of the parent account provided that the account must not be the root or the PCM login account. Otherwise, lock the object as is.

> **Note:**
>
> When using either flag, be aware of what is locked and what is not locked. A deadlock can occur when two different transaction contexts lock on a common set of objects and the lock sequences are not synchronized.

Oracle can detect the database-level deadlocks and throws an error to the DM. Users may also dump the CM lock map to determine the object that caused the deadlock. The technique is as follows:

1. Obtain the lock map flist using PCP_GET_TRANS_FLIST.
2. Use PIN_ERR_LOG_FLIST to display the lock map flist.

There are many opcodes that can lock based on the objects in the list. In some cases, it should be okay to leave things as they are. If you decide not to change anything, things should still work as before and performance should not degrade.

Some sample strategies that may be employed to assist performance are:

- Change PCM_OPFLG_LOCK_OBJ to PCM_OPFLG_LOCK_ DEFAULT in an operation you already use. This may change its locking behavior but typically if you were not experiencing deadlocks with this operation before, it can be changed safely.
- Change the object to lock. If an opcode works on only the **/service** object, try using the PCM_OPFLG_LOCK_DEFAULT flag on the **/service** object if it is provided in the input flist. In this way, objects that are not changed are not locked and therefore a given opcode will not interfere with objects that it does not change.

You can lock only when you are in a read-write transaction. Otherwise, your lock flags will be ignored or may result in a fatal PCM error.

Disabling Granular Object Locking

By default, granular object locking is enabled in BRM. Performance is better when you use granular object locking.

To disable granular locking, run the **pin_bus_params** utility to change the **LockConcurrency** business parameter. For information about this utility, see "pin_bus_params".

The default behavior, which corresponds to the **LockConcurrency** value of **high**, allows you to call opcodes with the PCM_OPFLG_LOCK_OBJ and PCM_OPFLG_LOCK_DEFAULT flags to determine the locking procedure. See "Locking Specific Objects".

When you disable granular locking, the following rules apply:

- There is no differentiation of the PCM_OPFLG_LOCK_OBJ and PCM_OPFLG_LOCK_DEFAULT flags.
- A given balance group object is locked as is.
- All other objects are translated to lock its associated account (if any).
- The root login account will be ignored.
- The object that has no account association will be locked as is.

To disable granular object locking:

1. Go to *BRM_home*/**sys/data/config**.

2. Create an XML file from the **/config/business_params** object:

   ```
   pin_bus_params -r multi_bal financial/config/xml_utils/bus_params_multi_bal.xml
   ```

3. In the file, change **high** to **normal**:

   ```
   <LockConcurrency>high</LockConcurrency>
   ```

4. Save the file as **financial/config/xml_utils/bus_params_multi_bal.xml**..
5. Load the XML file into the BRM database:

   ```
   pin_bus_params financial/config/xml_utils/bus_params_multi_bal.xml
   ```

6. Stop and restart the CM.

5

Understanding the PCM API

Learn about the Oracle Communications Billing and Revenue Management (BRM) Portal Communication Module (PCM) Application Programming Interface (API), which you use to interact with the BRM database.

Topics in this document:

- About the PCM API
- Header Files
- About Opcode Usage
- About Transaction Usage
- Calling PCM Opcodes
- Manipulating Objects in Custom Applications
- Supporting an Older Version of BRM

For information about the PIN (Portal Information Network) libraries, which you use to handle errors and to manipulate flists, POIDs, fields, strings, and decimal data types, see "PIN Libraries Reference" in *BRM Developer's Reference*.

About the PCM API

All access to the data in the BRM database is through the PCM API. Client applications and custom Facilities Modules (FMs) use this library to manipulate objects in the database.

The API consists of three classes of functions:

- Context management. You use context management opcodes to control communication channels to the database.
- Basic object manipulation. You use base opcodes to create, search for, delete, and modify objects in the database.
- FM object manipulation. You use FM opcodes to implement business policies and processes

You make BRM API calls by using a macro interface instead of directly through functions. When an API macro is called, the macro records the file name and line number of the source code where the API was called. If an error occurs, the macro logs a message including the file name and source code line number, making it easy to locate and correct the error.

For details on the return status of PCM functions and the error messages returned, see "Finding Errors in Your Code".

The API definitions are independent of the underlying storage model. The C data structures are opaque, and the opcodes are designed to appear much like object methods.

Context Management Opcodes

The context management opcodes open and close a communication channel to the BRM database by opening and closing a context to the Connection Manager (CM). The context structure is opaque to the application. It contains state data used by the PCM library to manage the communication channel.

The context management opcodes include functions for synchronous and asynchronous transactions. All transactions must follow these rules:

- Each application can connect to only one CM at a time.
- Only one connection can be open to a DM at a time.
- All object manipulation functions performed within a transaction must apply to the same BRM database schema.

When a context is opened, you can call additional functions to open, close, and commit or cancel transactions within the open context.

When you open a PCM context, a connection is established between your application and the BRM server. This connection adds significant overhead to the system because of the security and auditing checks performed by BRM. Therefore, to maximize performance, make sure your application keeps the context open until all the operations are performed. If your application opens and closes contexts frequently, performance will be affected.

If you are writing applications, such as Web-based Active Server Pages or CGI scripts, that cannot maintain an open context for a long time, use CM Proxy. CM Proxy allows your application to access the database with a pre-authorized connection and avoid the system overhead of a login for each connection.

For more information on CM Proxy, see "Using CM Proxy to Allow Unauthenticated Log On" in *BRM System Administrator's Guide*.

For more information on context management opcodes, see "Context Management Opcodes" in *BRM Opcode Guide*.

Base Opcodes

You use base opcodes to perform operations such as creating and manipulating objects, searching, and transaction handling. Base opcodes are implemented in the Data Manager, unlike the other opcodes, which are implemented in the Connection Manager.

Base opcodes require an open communication context and an input flist as parameters. The input flist specifies the input field arguments, and is not modified during execution.

You call the object manipulation opcodes with PCM_OP. The opcode you want to call is an input parameter.

Base opcodes pass back a return flist as a parameter. The return flist contains the result field arguments. The memory for the return flist is dynamically allocated.

You can run basic object manipulation macros in any combination within a transaction, depending on the resources available.

Search and Global Search Opcodes

Some base opcodes are used for searching in BRM. "Searching" in this context means looking in your BRM database for objects that meet a criteria that you specify. That is, you want the

POIDs of all the objects that share certain characteristics. The PCM_OP_SEARCH_* and PCM_OP_GLOBAL_SEARCH_* opcodes are designed for this purpose. They search single or multiple database schemas for accounts that match the criteria you specify and return the POIDs of those accounts.

After you know the POIDs of accounts, you can call other base opcodes designed to read or change data, such as PCM_OP_READ_OBJ, PCM_OP_WRITE_OBJ, or PCM_OP_DELETE_OBJ.

See "Searching for Objects in the BRM Database" for a discussion of searching and a list of the SEARCH opcodes. This document explains the BRM searching strategy, including the types of searching that BRM does by default and what you must know to write custom applications to use on the BRM database.

When you write a custom DM, depending on your needs, you implement opcodes from the following set:

- Base opcodes for LDAP DM. See "LDAP Base Opcodes" in *BRM Opcode Guide*.
- Base opcodes for Email DM. See "Email Data Manager Opcodes" in *BRM Opcode Guide*.

FM Opcodes

FMs (Facilities Modules) are shared libraries that implement higher-level opcodes. Each FM implements a set of opcodes to perform operations specific to that module. FMs create online accounts, manage customer-related information, charge customers for usage, and allow third-party systems to be integrated with BRM. In the billing FM, for example, opcodes perform advanced billing-related operations on user accounts.

You call the FM opcodes using PCM_OP, with the FM opcode you are calling as the input parameter.

FM opcodes are divided into the following types:

- **Standard FM opcodes** perform specific BRM operations. You cannot change the standard opcodes. However, to add new functionality, you can write new opcodes.

  For more information, see "Writing a Custom Facilities Module".

- **Policy FM opcodes** contain the BRM business logic. You can modify the default behavior of policy opcodes to suit your business needs. BRM includes the source code for all the policy opcodes.

  For example, you can bill customers on their anniversary date or on the first day of each month by modifying the default implementation of the PCM_OP_CUST_POL_PREP_ACTINFO policy opcode.

  For more information on customizing policy opcodes, see "About System and Policy Opcodes".

BRM includes a set of policy opcodes, including source code, as hooks for you to add your code. These opcodes do not have a default implementation.

Standard FMs that use business logic to process requests have policy FMs associated with them. A few FMs, such as the SDK FM and the Group FM, which are internal to BRM and do not need business logic for processing data, do not have associated policy FMs.

Each of the BRM optional managers has its own FM. See the appropriate optional manager documentation for more information.

About the PREP and VALID Opcodes

Many opcodes, for example, PCM_OP_CUST_SET_LOGIN and PCM_OP_CUST_PREP_CUSTOMER, call policy PREP and VALID opcodes, such as PCM_OP_CUST_POL_PREP_PASSWD and PCM_OP_CUST_POL_VALID_PASSWD. You can use PREP and VALID opcodes to customize how data is processed.

- Use the PREP opcodes to process data before it is validated. Typical processing includes adding missing fields whose values are derived or generated by the PREP operation, and forcing fields to predefined values independent of what the customer specified. PREP opcodes are given a set of customer-specified fields on the input flist, and return the processed version of the same data on the output flist.

  If a PREP opcode cannot derive all the necessary fields because the customer-specified values used in the derivation are incorrect, no error is returned. Instead, the derived fields are put on the output flist with a default value, and the corresponding VALID call detects the incorrect data and returns the validation error to the calling application. This approach allows the calling application to see the details of the validation error rather than receiving a less precise **ebuf** error passed up from the PREP opcode.

  If a PREP opcode cannot generate a necessary field or some other internal problem is encountered, an **ebuf** error is returned.
- Use the VALID opcodes to validate field values. Typical checks include formatting tests for data integrity, tests for illegal values and tests for required information that is missing. VALID opcodes are given a set of related fields and values on the input flist, and return a list of fields that failed the validation tests on the output flist. The VALID opcodes cannot alter the value of a field that is not suitable, that is the purpose of the PREP opcodes.

  If one or more fields fail the validation tests, they are returned using the PIN_FLD_FIELDS array on the output flist. This array is structured to allow fields nested within arrays or substructs to be accurately represented. All fields that failed validation are returned by the operation, so the caller can correct all errors at once and retry the operation.

Validating Fields by Using Field Validation Editor

To validate the fields that you specify in the Field Validation Editor, use PCM_OP_CUST_VALID_FLD.

Header Files

Each set of related opcodes has a corresponding header file. Your custom code and applications must include the header files that correspond to the opcodes you use.

Context management opcodes use the **pcm.h** header file. Always include this file in your applications.

Header files for base opcodes and Facilities Module (FM) opcodes are located in the **include/ops** directory. To include one of these header files, use this syntax:

```
#include "ops/file.h"
```

Where *file* is the name of the header file.

For example, if your application calls PCM_OP_CUST_COMMMIT_CUSTOMER, you must include the **ops/cust.h** header file.

About Opcode Usage

Recommended: opcodes are designed specifically for you to call from your custom applications. They are not expected to change from release to release.

Limited: opcodes should only be called in special cases. They may change from release to release.

Last Resort: opcodes should only be called if absolutely necessary. Calling these opcodes means that you are either on the wrong track or rewriting major portions of code. BRM will change these opcodes as necessary.

About Transaction Usage

A transaction is a connection that requires that the data being read or written must not change during the connection. A transaction adheres to the “ACID" properties, which means the transaction is:

- Atomic: Either the entire transaction completes successfully, or none of it does.
- Consistent: The transaction takes the database from one consistent state to another.
- Isolated: Only the process that opened the transaction can see the intermediate results of the transaction.
- Durable: After the transaction is committed to the database, the changes are permanent and cannot be changed except by another transaction.

Each opcode uses one of the following types of transaction handling:

- Transaction Handling: Required
- Transaction Handling: Requires New
- Transaction Handling: Supports

Transaction Handling: Required

The transaction for this opcode can be wrapped in a transaction opened by another opcode.

If a read-write transaction is already open when this opcode is run, all data modifications take place within the open transaction. The modifications are committed or cancelled along with all other changes when the transaction is committed or cancelled.

If no transaction is open when the opcode is called, a read-write transaction is opened. All actions are performed within this transaction, ensuring that the entire operation is performed atomically. If an error occurs during the execution of the opcode, all changes are cancelled when the transaction is cancelled. If no error occurs, the transaction is committed at the end of the operation.

This opcode requires a read-write transaction. It is therefore an error to have a read-only transaction open when this opcode is called.

Transaction Handling: Requires New

This opcode manages transactions internally to ensure absolute integrity of the database. A transaction for this opcode cannot be wrapped in another transaction.

If no transaction is open when the opcode is called, a read-write transaction is automatically opened and all actions are performed within this transaction.

If a transaction is already open when the opcode is called, an error occurs.

Transaction Handling: Supports

This opcode does not modify object data. If it is called while a transaction is not already open, the operation is run without transactional control.

If a read-write or read-only transaction is already open when this opcode is called, the opcode is run as part of the transaction and reads the in-process state of the data.

If the opcode is called when a separate, unrelated transaction is taking place, it reads the last saved state of the database.

Calling PCM Opcodes

You call the base and FM opcodes by using **PCM_OP()**. You pass the opcode you want to call as one of the input parameters. **PCM_OP()** runs the opcode in its input parameters list in an open communication channel or a context.

You use the following parameters and flags with **PCM_OP()**:

pcm_ctxp

- Pointer to an open PCM context.

opcode

- Name of the opcode you want to call.

flags

- **(int32)NULL**

  No flags specified. Use only when there are no flags defined for this operation.

- **PCM_OPFLG_READ_RESULT**

  Returns all the fields in the object from the output flist, not just the POID. Valid only for opcodes that create objects.

- **PCM_OPFLG_CALC_ONLY**

  Calculate only. Valid only for opcodes that create objects. No fields in the database are changed and the object is not actually created. Instead, fields that would have been used to create the object are returned to the caller on the output flist.

in_flistp

- An input flist specification for the opcode defining the required and optional input fields for the opcode to function properly. Each opcode has an input flist specification that you must use to create the input. See the input flist specification in an individual opcode description for details.

ret_flistpp

- An output flist specification defining what you expect the opcode to return. Each opcode has an output flist specification that you must use to create the input. You must explicitly destroy the return flist to free memory. See the output flist specification in an individual opcode description for details.

ebufp

- Pointer to an error buffer. Used to pass status information back to the caller.

The following example shows how to call the policy opcode PCM_OP_CUST_POL_GET_PLANS to get packages:

```
/*Declarations*/
pin_errbuf_t *ebufp
input_flistp = PIN_FLIST_CREATE(ebufp);
return_flistp = PIN_FLIST_CREATE(ebufp);

PCM_OP(ctxp, PCM_OP_CUST_POL_GET_PLANS, 0, input_flistp, &return_flistp, ebufp);
```

Manipulating Objects in Custom Applications

There are three basic steps to manipulating objects in a custom application or module:

1. Open a context by calling PCM_CONTEXT_OPEN or PCM_CONNECT in an application.
2. Call opcodes with PCM_OP.
3. Close the context by using PCM_CONTEXT_CLOSE.

Supporting an Older Version of BRM

The PIN_FLD_VERSION flag on the PCM_OP_CUST_COMMIT_CUSTOMER input flist specifies whether the flist complies with the current version of BRM. If it doesn't, new BRM objects are created. This supports backward compatibility. The current version is this version of BRM.

Possible flag values are:

- PIN_PORTAL_VERSION_CURRENT (a value of **0**) specifies this version of BRM.
- PIN_PORTAL_VERSION_LEGACY (a value of **1**) specifies a legacy version of BRM. This is the default. If PIN_PORTAL_VERSION_LEGACY is specified, the input flist is converted to the current version and all necessary objects, including the **/billinfo**, **/balance_group**, and **/payinfo** objects are created for the account.

6

Accessing Configuration Files and Objects in Custom Code

Learn how to access **pin.conf** files and **/config/business_params** objects in your custom Oracle Communications Billing and Revenue Management (BRM) code.

Topics in this document:

- Accessing pin.conf Files in Custom Code
- Using /config/business_params Objects

Accessing pin.conf Files in Custom Code

You can use the PCM C++ **PinConf** class to enable your code to read values from a **pin.conf** file. For example, this code is from the policy source file **fm_rate_pol_tax_loc.c**. This code gets the value of the customer's tax locale from the Connection Manager (CM) **pin.conf** file:

```
/*******************************************************
 * Look up the ISP city from pin.conf
 *******************************************************/

pin_conf("fm_rate_pol", "provider_loc", PIN_FLDT_STR,
(caddr_t *)&locale, &perr);
```

The entry in the **pin.conf** file looks like this:

```
#========================================================================
# provider_loc
#
# City, state, ZIP code, and country where you provide services to
# your customers.
#
# This information is used to determine tax rates.
#========================================================================
- fm_rate_pol provider_loc Cupertino, CA 95014 USA
```

In the following example, this code in the **fm_subscription_pol_spec_cancel.c** policy source code file gets a value (0 or 1) from an entry in the CM **pin.conf** file:

```
/* Find all charge offers without a provisioning tag; cancel and
 * delete charge offer from table.
 */

if (pin_conf_keep_cancelled_products_or_discounts == 0){
    PIN_FLIST_FLD_SET(p_arrayp, PIN_FLD_ACTION,
    PIN_BILL_CANCEL_PRODUCT_ACTION_CANCEL_DELETE, ebufp);
} else {
    PIN_FLIST_FLD_SET(p_arrayp, PIN_FLD_ACTION,
    PIN_BILL_CANCEL_PRODUCT_ACTION_CANCEL_ONLY, ebufp);
}
```

The following example shows the entry in the **pin.conf** file.

```
#=========================================================================
# keep_cancelled_products_or_discounts
#
# Specifies whether to keep canceled charge offers and discount offers
# associated with a specified account.
#
# The value for this entry can be one of the following:
#
# 1 = (Default) Deletes the canceled charge offer (/purchased_product [# object) or
discount offer (/purchased_discount object) by using
# PCM_OP_DELETE_OBJ.
#
# 0 = Keeps the canceled /purchased_product or /purchased_discount object
# and sets its STATUS field to PIN_PRODUCT_STATUS_CANCELLED or
# PIN_DISCOUNT_STATUS_CANCELLED respectively.
#
#=========================================================================
- fm_subscription_pol keep_cancelled_products_or_discounts 1
```

In addition to retrieving a value from a **pin.conf** file, you can hard code a default value that is used if the **pin.conf** entry is not present.

For information about the **PinConf** class, see "Accessing Configuration Values by Using pin.conf".

Using /config/business_params Objects

You can customize BRM by adding new business parameters to control various aspects of BRM operations and calling these business parameters from policy opcodes. You can also add completely new business parameter classes to BRM.

Adding and Loading New Parameters

Adding parameters is useful if you are customizing existing functionality; for example, to expand the criteria used to determine whether a payment should be suspended. To do this, you customize PCM_OP_PYMT_POL_VALIDATE_PAYMENT, the policy opcode that validates payments, to filter any payments below a specified amount.

For added flexibility, you may also want the ability to turn off this filter at certain times. One way to do this is to add a parameter to the **/config/business_params** object for the **ar** parameter class and have PYMT_POL_VALIDATE_PAYMENT check that parameter.

To implement the **/config/business_params** part of this process, you create a new parameter that you enable or disable depending on whether you want to filter payments below a specified amount so that these payments do not get suspended. This parameter will be called **payment_suspense_amount_filter** in the **/config/business_params** object and **PaymentSuspenseAmntFilter** in the supporting XML file set. You add the parameter as follows:

1. Modify the **bus_params_AR.xsd** file in the *BRM_home***/sys/data/config/** directory to add the new parameter. (*BRM_home* is the directory in which the BRM server software is installed.)

```
<xs:element name="PaymentSuspenseAmntFilter" type="switch">
  <xs:annotation>
    <xs:documentation xml:lang="en">Enable/Disable filtering
    of payment suspense based on payment amount. The parameter
    values can be 0 (disabled) or 1 (enabled). The default is 0
    (disabled).</xs:documentation>
```

```
  </xs:annotation>
</xs:element>
```

2. Modify the **bus_params_AR.xsl** file in the *BRM_home*/**sys/data/config/** directory to add the new parameter:

```
<xsl:template match="bc:PaymentSuspenseAmntFilter">
  <xsl:element name="Param">
    <xsl:element name="Name">
      <xsl:text>payment_suspense_amount_filter</xsl:text>
    </xsl:element>
    <xsl:element name="Desc">
      Enable/Disable filtering of payment suspense based on
      payment amount. The parameter values can be 0 (disabled)
      or 1 (enabled). The default is 0(disabled).
    </xsl:element>
    <xsl:element name="Type">INT</xsl:element>
    <xsl:element name="Value">
      <xsl:choose>
        <xsl:when test="text() = 'enabled'">
          <xsl:text>1</xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>0</xsl:text>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:element>
</xsl:template>
```

3. Modify the **bus_params_to_AR.xsl** file in the *BRM_home*/**sys/data/config/** directory to add the new parameter:

```
<xsl:when test="$name = 'payment_suspense_amount_filter'">
  <xsl:element name="PaymentSuspenseAmntFilter">
    <xsl:choose>
      <xsl:when test="$value = '1'">
        <xsl:text>enabled</xsl:text>
      </xsl:when>
      <xsl:when test="$value = '0'">
        <xsl:text>disabled</xsl:text>
      </xsl:when>
    </xsl:choose>
  </xsl:element>
</xsl:when>
```

4. Use the **pin_bus_params** utility to retrieve the **ar** instance of the **/config/business_params** object:

```
pin_bus_params -r BusParamsAR bus_params_AR.xml
```

5. Modify the resulting XML file to add the new parameter:

```
<PaymentSuspenseAmntFilter>disabled</PaymentSuspenseAmntFilter>
```

6. Use the **pin_bus_params** utility to load the object from the modified XML file:

```
pin_bus_params bus_params_AR.xml
```

For information on using the **pin_bus_params** utility, see "pin_bus_params".

Adding and Loading New Parameter Classes

You might need to add parameter classes when you create BRM features or customize existing functionality that has no associated parameter class.

> **Note:**
>
> You can determine if BRM has a parameter class for a certain functionality by looking at the support files. For example, **bus_params_AR.xml** and **bus_params_billing.xml** indicate that there are parameter classes for accounts receivable and billing. Support files for parameter classes are located in *BRM_home***/xsd.**

To create a parameter class, you create the following files:

- XML file: **bus_params**_parameter_class_name**.xml**: Contains the parameter settings from the **/config/business_params** object for the parameter class. Parameter settings in this file are loaded into the object by using the **pin_bus_params** utility. This file is located in *BRM_home***/sys/data/config**.
- XSD file: **bus_params**_parameter_class_name**.xsd**: Validates the contents of the **bus_params**_parameter_class_name**.xml** file when loading the object. This file is located in *BRM_home***/xsd**.
- XSL file: **bus_params**_parameter_class_name**.xsl**: Translates the contents of the **bus_params**_parameter_class_name**.xml** file into the correct format for the **/config/business_params** object. The **pin_bus_params** utility calls this file when loading the object. This file is located in *BRM_home***/xsd**.
- XML translation file: **bus_params_to**_parameter_class_name**.xsl**: Translates the contents of the **/config/business_params** object into XML format during object retrieval. This file is located in *BRM_home***/xsd**.

For example, if you used custom policy opcodes to create a rewards tracking application, you could use a business parameter to switch between tracking frequent flier miles and tracking minutes. To do so, you would create the following:

- A parameter class. In the **/config/business_params** object, the class would be named **rewards**. In the XML files, the class would be named **BusParamsRewards**.
- A business parameter named **RewardsTracking**.To track miles, the option would be set to **0**, to track minutes, it would be set to **1**.

  This business parameter would be named **rewards-tracking** in the **/config/business_params** object. The policy opcode would use the value of **rewards-tracking**.

To support the new parameter class and business parameter, you would create these files:

- **bus_params_rewards.xml**
- **bus_params_rewards.xsd**
- **bus_params_rewards.xsl**
- **bus_params_to_rewards.xsl**

To create these files:

1. Copy one of the **bus_params**_parameter_class_name**.xsd** sample files in *BRM_home***/xsd** and save it as **bus_params_rewards.xsd**. Modify the file as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema   targetNamespace="http://www.portal.com/schemas/BusinessConfig"
  xmlns:businessConfig="http://www.portal.com/schemas/
    BusinessConfig"
```

```
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:annotation>
    <xs:documentation xml:lang="en"
    </xs:documentation>
  </xs:annotation>

  <xs:complexType name="BusParamsRewardsType">
    <xs:sequence>
      <xs:element name="RewardsTracking" type="switch">
        <xs:annotation>
          <xs:documentation xml:lang="en">
            The reward to track. The parameter values can be
            0 (Miles) or 1 (Minutes). The default is
            1 (Minutes).
          </xs:documentation>
        </xs:annotation>
        <xs:simpleType name="rewardtrack">
          <xs:restriction base="xs:string">
            <xs:enumeration value="Miles" />
            <xs:enumeration value="Minutes" />
            <xs:whiteSpace value="collapse" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

2. Copy one of the **bus_params**_parameter_class_name_**.xsl** sample files in *BRM_home*/**xsd** and save it as **bus_params_rewards.xsl**. Modify the file as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>

<xsl:stylesheet
  version="1.0"   xmlns="http://www.portal.com/schemas/BusinessConfig"
  xmlns:bc="http://www.portal.com/schemas/BusinessConfig"   xmlns:xsl="http://
www.w3.org/1999/XSL/Transform"   exclude-result-prefixes="bc">

  <xsl:output method="xml" indent="yes" />

  <xsl:template match="/">
    <BusinessConfiguration       xmlns="http://www.portal.com/schemas/
BusinessConfig"       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.portal.com/schemas/
        BusinessConfig business_configuration.xsd">
      <BusParamConfiguration>
      <BusParamConfigurationList>
        <ParamClass desc="Business logic parameters for Reward Tracking"
name="rewards-tracking">
          <xsl:apply-templates            select="/bc:BusinessConfiguration/
            bc:BusParamConfigurationClass/
            bc:BusParamsRewardsType/bc:*" />
        </ParamClass>
      </BusParamConfigurationList>
      </BusParamConfiguration>
    </BusinessConfiguration>
  </xsl:template>

  <xsl:template match="bc:RewardsTracking">
```

```
    <xsl:element name="Param">
      <xsl:element name="Name">
        <xsl:text>rewards-tracking</xsl:text>
      </xsl:element>
      <xsl:element name="Desc">
        The reward to track. The parameter values can be
        0 (Miles) or 1 (Minutes). The default is
        1 (Minutes).
      </xsl:element>
      <xsl:element name="Type">INT</xsl:element>
      <xsl:element name="Value">
        <xsl:choose>
          <xsl:when test="text() = 'Minutes'">
            <xsl:text>1</xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:text>0</xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:element>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

3. Copy one of the **bus_params_to_***parameter_class_name***.xsl** sample files in *BRM_home*/**xsd** and save it as **bus_params_to_rewards.xsl**. Modify the file as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>

<xsl:stylesheet
  version="1.0"   xmlns="http://www.portal.com/schemas/BusinessConfig"
  xmlns:bc="http://www.portal.com/schemas/BusinessConfig"   xmlns:xsl="http://
www.w3.org/1999/XSL/Transform"   exclude-result-prefixes="bc">

  <xsl:output method="xml" indent="yes" />

  <xsl:template match="/">
    <BusinessConfiguration       xmlns="http://www.portal.com/schemas/
BusinessConfig"       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.portal.com/schemas/
        BusinessConfig business_configuration.xsd">
      <BusParamConfigurationClass>
        <BusParamsRewards>
          <xsl:apply-templates             select="/bc:BusinessConfiguration/
            bc:BusParamConfiguration/bc:BusParamConfigurationList/
            bc:ParamClass/bc:Param" />
        </BusParamsRewards>
      </BusParamConfigurationClass>
    </BusinessConfiguration>
  </xsl:template>

  <xsl:template match="//bc:Param">
    <xsl:variable name="name">
      <xsl:value-of select="bc:Name/text()" />
    </xsl:variable>
    <xsl:variable name="value">
      <xsl:value-of select="bc:Value/text()" />
    </xsl:variable>
    <xsl:choose>
      <xsl:when test="$name = 'rewards-tracking'">
        <xsl:element name="RewardsTracking">
          <xsl:choose>
```

```
            <xsl:when test="$value = '1'">
              <xsl:text>Minutes</xsl:text>
            </xsl:when>
            <xsl:when test="$value = '0'">
              <xsl:text>Miles</xsl:text>
            </xsl:when>
          </xsl:choose>
        </xsl:element>
      </xsl:when>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

4. Copy one of the **bus_params**_*parameter_class_name*.**xml** sample files in *BRM_home*/**sys/data/config** and save it as **bus_params_rewards.xml**. Modify the file as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<BusinessConfiguration      xmlns="http://www.portal.com/schemas/BusinessConfig"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http:2www.portal.com/schemas/Business
     Config business_configuration.xsd">

  <BusParamConfigurationClass>
    <BusParamsRewards>
      <RewardsTracking>
        Minutes
      </RewardsTracking>
    </BusParamsRewards>
  </BusParamConfigurationClass>

</BusinessConfiguration>
```

5. Modify the **bus_params_conf.xsd** file in the *BRM_home*/**xsd** directory to add the new parameter class.
   a. Add the following line to the schema location segment of the file:

      ```
      <xs:include schemaLocation="bus_params_rewards.xsd"/>
      ```

   b. Add the following line to the parameter class selection segment of the file:

      ```
      <xs:element name="BusParamsRewards" type="BusParamsRewardsType"/>
      ```

6. Use the **pin_bus_params** utility to load the **bus_params_rewards.xml** file:

   ```
   pin_bus_params bus_params_rewards.xml
   ```

When the new parameter class is created, and the business parameter is loaded, you can modify the custom policy opcode to look for the **rewards-tracking** value in the **/config/business_params** object.

Examples of Accessing Business Parameters in Custom Code

Opcodes read configuration values from **/config/business_params** objects to determine whether to run various functions. The following examples show how several BRM policy opcodes call values from **/config/business_params** objects.

Calling Business Parameters from PCM_OP_PYMT_POL_VALIDATE_PAYMENT

In its default implementation, the PCM_OP_PYMT_POL_VALIDATE_PAYMENT policy opcode checks whether payment suspense management is enabled. If so, it places payments that could not be validated into suspense.

This code in the **fm_pymt_pol_validate_payment.c** policy source file determines whether to suspend payments that can't be validated. To make this determination, BRM calls the **psiu_bparams_get_int()** function and uses the **psiu_business_params.h** header file to retrieve specific parameters from the appropriate **/config/business_params** object. This information is used to determine whether payment suspense management is enabled (PSIU_BPARAMS_AR_PYMT_SUSPENSE_ENABLED):

```
/*********************************************************
* Check if Payment Suspense Management feature is enabled
*********************************************************/
pymt_suspense_flag = psiu_bparams_get_int(ctxp, PSIU_BPARAMS_AR_PARAMS,
     PSIU_BPARAMS_AR_PYMT_SUSPENSE_ENABLE, ebufp);
if ((pymt_suspense_flag != PSIU_BPARAMS_AR_PYMT_SUSPENSE_ENABLED)&&
     (pymt_suspense_flag != PSIU_BPARAMS_AR_PYMT_SUSPENSE_DISABLED))
     {
       pin_set_err(ebufp, PIN_ERRLOC_FM,
         PIN_ERRCLASS_SYSTEM_DETERMINATE,
         PIN_ERR_INVALID_CONF, 0, 0, 0);
       PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_ERROR,
       "bad param value for \"payment_suspense_enable\" in /config/business_params",
         ebufp);
     }
```

The segment that enables payment suspense management in the **/config/business_params** object looks like this:

```
0 PIN_FLD_PARAMS          ARRAY [2] allocated 4, used 4
1    PIN_FLD_DESCR          STR [0] "Enable/Disable payment suspense management.
                                     The parameter values can be 0 (disabled),
                                     1 (enabled). Default is 0 (disabled)."
1    PIN_FLD_PARAM_NAME     STR [0] "payment_suspense_enable"
1    PIN_FLD_PARAM_TYPE     INT [0] 1
1    PIN_FLD_PARAM_VALUE    STR [0] "1"
```

Calling Business Parameters from PCM_OP_BILL_POL_REVERSE_PAYMENT

In its default implementation, the PCM_OP_BILL_POL_REVERSE_PAYMENT policy opcode reverses payments applied to accounts that were written off; it does not reverse the payment if the write-off reversal was anything other than an account-level write-off.

This code in the **fm_bill_pol_reverse_payment.c** policy source code determines whether the write-off was at the account level. To make this determination, BRM calls the **psiu_bparams_get_str()** function and uses the **psiu_business_params.h** header file to retrieve specific parameters from the appropriate **/config/business_params** object. This information is used to determine whether write-off level is PSIU_BPARAMS_AR_PYMT_SUSPENSE_ENABLED. If so, it reverses the payment, again writing off the account:

```
/*********************************************************
* Verify if write off level set to "a" (account) in
* /config/business_params and Call PCM_OP_AR_ACCOUNT_WRITEOFF
*********************************************************/
```

```
psiu_bparams_get_str(ctxp, PSIU_BPARAMS_AR_PARAMS,
PSIU_BPARAMS_AR_WRITEOFF_LEVEL, writeoff_rev_level, 2, ebufp);

if ( status_flag && ( *status_flag == PIN_PYMT_WRITEOFF_SUCCESS ) &&
     writeoff_rev_level &&
     !strcmp(writeoff_rev_level, PIN_WRITEOFF_REV_LEVEL_ACCOUNT) )
     {
       i_flistp = PIN_FLIST_CREATE(ebufp);

       vp = PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_POID, 0, ebufp);
       PIN_FLIST_FLD_SET(i_flistp, PIN_FLD_POID, vp, ebufp);

       vp = PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_PROGRAM_NAME, 0, ebufp);
       PIN_FLIST_FLD_SET(i_flistp, PIN_FLD_PROGRAM_NAME, vp, ebufp);
       vp = PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_START_T, 1, ebufp);
       if (vp)
       {
         PIN_FLIST_FLD_SET(i_flistp, PIN_FLD_START_T,
         (void *) vp, ebufp);
       }
```

The segment that determines the write-off level in the **/config/business_params** object looks like this:

```
0 PIN_FLD_PARAMS          ARRAY [2] allocated 4, used 4
1    PIN_FLD_DESCR          STR [0] "Selection of level of writeoff to be tracked for the
                                     purpose of writeoff reversal. Values can be
a(Account),
                                     b(Bill), i(Item), *(Any)."
1    PIN_FLD_PARAM_NAME     STR [0] "writeoff_level"
1    PIN_FLD_PARAM_TYPE     INT [0] 5
1    PIN_FLD_PARAM_VALUE    STR [0] "a"
```

7

Understanding the BRM Data Types

Learn about the data types that Oracle Communications Billing and Revenue Management (BRM) supports. The data types described here are defined in the **pcm.h** file.

Topics in this document:

- About the BRM Data Types
- Simple Data Types
- Portal Object ID (POID)
- Decimal Data Type
- Arrays
- Substructure
- Buffer Data
- Error Buffer

About the BRM Data Types

BRM supports a set of data types that you use to define fields in a storable class or in field lists (flists). For information on flists and storable classes, see "Understanding Flists".

Table 7-1 lists the data types that BRM supports. Some of the BRM data types are simple data types, which map to data types in programming languages such as C and C++. The others hold more complex data and point to C structures as their value. The complex data types that are specific to BRM or used in a special way in BRM, such as the Portal object ID (POID), arrays, and substructs, are explained in detail in the following sections.

Table 7-1 BRM Supported Data Types

Data Type	Description	C Value
PIN_FLDT_INT	Signed 32-bit integer. Contains four bytes of data represented by a number. BRM considers an integer value that begins with 0 as octal, and an integer value that begins with 0x as hexadecimal and converts the value into decimal.	**int32**
PIN_FLDT_ENUM	Enumerated value. Contains a list of well-known values.	**enum**
PIN_FLDT_DECIMAL	Decimal data type, number of decimal places determined by MAX.	**pin_decimal_t**
PIN_FLDT_STR(len)	ASCII character string terminated with a **\0** (NULL). **len** = max length in bytes, not including **\0**. It uses UTF-8 encoding.	**char ***
PIN_FLDT_BINSTR(len)	A string of binary data. **len** = max length in bytes.	**pin_binstr_t**

Table 7-1 (Cont.) BRM Supported Data Types

Data Type	Description	C Value
PIN_FLDT_TSTAMP	Linux timestamp with one second accuracy. Contains integer data. This number is interpreted as the number of seconds past January 1, 1970.	**time_t**
PIN_FLDT_POID	Portal object identifier. See "Portal Object ID (POID)".	**poid_t ***
PIN_FLDT_ARRAY	Array element. See "Arrays".	**pin_flist_t ***
PIN_FLDT_SUBSTRUCT	Embedded substructure. See "Substructure".	**pin_flist_t ***
PIN_FLDT_BUF	Buffer with an arbitrary size of any large data such as text, image, or any other kind of data.	**pin_buf_t**
PIN_FLDT_ERRBUF	Structure for error holding error information.	**pin_errbuf_t**

Simple Data Types

BRM supports the following simple data types that map to data types in the C programming language:

- PIN_FLDT_INT
- PIN_FLDT_ENUM
- PIN_FLDT_DECIMAL
- PIN_FLDT_STR(len)
- PIN_FLDT_BINSTR(len)
- PIN_FLDT_TSTAMP

See Table 7-1 for the C values.

Portal Object ID (POID)

The POID data type identifies an object in the BRM database. Each object has a unique POID in BRM. You use the POID to locate an object in the database.

Use the POID management macros in the PIN Library to manipulate the POIDs.

The POID contains the following information:

database_number object_type object_id object_revision_level

Example:

```
0.0.0.1 /account 1234 0
```

You can specify a type-only POID to perform an action on all objects of a particular type. For example, to search for all **/device/sim** objects, the input flist for the search opcode would contain the following field:

```
1     PIN_FLD_POID           POID [0] 0.0.0.1 /device/sim -1 0
```

Table 7-2 describes each entry in the POID:

Table 7-2 POID Entries

Entry	Description
database number	An arbitrary 64-bit number assigned to a particular BRM database by the BRM system administrator. Each database has a unique database number that is stored in each object in that database. This number must be used by all programs, CMs, and DMs accessing that database. Decimal dotted notation is used for the database number: 0.0.0.*x*, where *x* is the database number, such as 1057. See the **dm_pointer** entry in the CM **pin.conf** file for an example.
object type	The storable class to which the object belongs, for example, **/event** and **/service.** Null poid has only **/.**
object id	A unique 64-bit number assigned to each object. Once assigned, the ID is never changed or re-used. The ID is a 64-bit number to accommodate the large number of objects that can exist within a single database. The ID is guaranteed to be unique within a given database, *not* across databases. The maximum value allowed for the ID in a nonpartitioned table is 2^{64} and in a partitioned table is 2^{44}.
object revision level	Revision number. This value is incremented automatically each time the object is updated. You cannot change this value directly.

Decimal Data Type

The decimal data type, PIN_FLDT_DECIMAL is an opaque data type that you use to represent values precisely to a specified number of decimal places.

You cannot perform arithmetic operations on a void pointer and C has no operator overloading. Therefore, the API provides a set of functions to perform arithmetic operations on **pin_decimal_t**.

The BRM C API uses **pin_decimal_t**, and it is defined in *BRM_home***/include/pin.h** file, where *BRM_home* is the directory in which the BRM server software is installed.

You manipulate the decimal data type, by using the decimal functions in the PIN libraries. You can perform the following arithmetic operations by using the decimal data type functions:

- Convert string to decimal and decimal to string
- Add, subtract, multiply, and divide two decimals
- Compare two decimals
- Scale or round a number
- Negate a decimal
- Output to a string
- Output to a double

For detailed descriptions of the functions, see Decimal Data Type Manipulation Functions in *BRM Developer's Reference*.

> **Note:**
>
> **PIN_FLDT_DECIMAL** replaces the data type **PIN_FLDT_NUM** from earlier releases.

Arrays

Use the array data type PIN_FLDT_ARRAY to store a defined structure of information. An array contains a recurring set of data structures called *elements*. Each element in an array can contain multiple fields, including other nested arrays. Each element in an array must contain the same *number* and *type* of fields as all the other elements in the array.

For example, the **/account** storable class contains an array called PIN_FLD_NAMEINFO. Each element in this array has fields for first name, last name, street address, and other address information. There can be any number of elements in the array to describe the different types of account addresses.

Each field in an array element has an element ID, which specifies the element of the array to which the field belongs. This element ID, in addition to the field name of the array, uniquely identifies the field in an object.

Arrays in BRM are sparse arrays and *not* C language style arrays. The elements in the array are not in any sequential order. Unlike C, the array element *a[24]* does not mean that there are 23 elements preceding it.You can add an element in any order with an arbitrary element ID.

The elements of a BRM array are *not* pre-allocated; they are assigned by applications as needed. Therefore, the missing elements in the sequence of element IDs do not use any memory or disk space.

You can add and delete elements from an array using the flist field manipulation macros. When you add an element to an array by using PIN_FLIST_ELEM_ADD, an array is automatically created. You do not have to create an array before adding elements to it.

For information on how to use the macros, seeFlist Field-Handling Macros in *BRM Developer's Reference*.

Substructure

Use the substruct data type PIN_FLDT_SUBSTRUCT to group several data types. You use substructs to define a field that contains several fields of different data types. Substructs can contain any of the BRM supported data types, including arrays, and they can be nested to any level.

> **Note:**
>
> Use substructs to create subclasses of the default storable classes included with BRM.

Because there is only one element, substructs are fully identified by the field name in the storable class. Unlike the arrays, they do not require element IDs.

Use the Flist field-handling macros to create and manipulate substructs.

Buffer Data

The buffer type flist field (PIN_FLDT_BUF) is used for large text files or binary data as an array of bytes.

xbuf stands for external buffer. The **buf** data is not in memory but is written directly from a file to the wire or from the wire to a file. The most common use for **xbufs** are for systems with limited or slow virtual memory. **xbufs** can only be used from an application.

A buffer (**buf**) field represented by the **pin_buf_t** has the following structure:

```
/*
      * data buffer.
      */
     typedef struct pin_buf {
            int32    flag;        /* if XBUF, ... */
            int32    size;        /* size of data */
            int32    offset;      /* offset (for read) */
            caddr_t  data;        /* pointer to data (BUF) */
            char     *xbuf_file;  /* ptr to filename for XBUF */
    } pin_buf_t;
```

xbuf values are defined for the flag field. These can be **bit-wise-OR**ed together.

```
/* users want data from/to a file...*/

#define PCM_BUF_FLAG_XBUF       0x0001

/* if XBUF, encode filename, not data*/

#define PCM_BUF_FLAG_XBUF_READ  0x0002
```

Table 7-3 describes the **pin_buf_t.flag** values.

Table 7-3 Values for pin_buf_t.flag

Flag	Description
pin_buf_t.flag = 0x0	Buffer data is assumed to be available in **pin_buf_t.data** field in memory. The **pin_buf_t.xbuf_file** field is ignored.
pin_buf_t.flag = 0x1 (PCM_BUF_FLAG_XBUF)	Use this to write data to a **buf** field in an object. The buffer data is assumed to be available in the file pointed to by the **pin_buf_t.xbuf_file** field. The data is read from the file only when the flist is shipped on the wire.
pin_buf_t.flag = 0x3 (PCM_BUF_FLAG_XBUF | PCM_BUF_FLAG_XBUF_READ)	Use this to read data from a **buf** field in an object. The buffer data is written directly to the file pointed to by the **pin_buf_t.xbuf_file** field.

Setting Buffer Data Fields in an Flist

The following example shows how to set a buffer field in an flist:

```
pin_buf_t buft;
buft.flag       = 0;
buft.size       = 26;
buft.offset     = 0; /* not used */
buft.data       = "abcdefghijklmnopqrstuvwxyz";
buft.xbuf_file  = NULL; /* not used */
```

```
PIN_FLIST_FLD_SET(flistp, PIN_FLD_BUFFER, &buft, &ebuf);
```

To avoid reallocating memory and copying the buffers in large buffers, use **PIN_FLIST_FLD_PUT**.

> **Note:**
>
> When you use **PUT** instead of **SET,** allocate memory on the heap for both the **pin_buf_t** data structure *and* the buffer data.

Getting Buffer Fields From an Flist

When accessing buffer fields from an flist, you set a pointer to a **pin_buf_t** data structure.

If you use **TAKE()** instead of **GET()**, make sure you free up the **pin_buf_t** structure, the data pointer, and the **xbuf_file** members in it.

Specifying Buffer Data Fields in Flist Converted to Strings

You can specify buffer fields in flist. For example, you might want to load an flist with a **buf** field into **testnap**. The following example provides the buffer data in place:

```
0 PIN_FLD_POID POID [0] $DB /xx 1
0 PIN_FLD_BUFFER BUF [0] flag/size/offset 0x2 26 0 data:
     0x000000 6162636465666768696a6b6c6d6e6f70
     0x000010 7172737475767778797a
```

You can specify **xbuf** data as in the following example, where the file *./xxx* is read and the contents sent to the wire:

```
>testnap
 ===> database 0.0.0.1 from pin.conf "userid"
> r xxx 1
> d 1
0   PIN_FLD_POID POID [0] $DB /xx 1
0   PIN_FLD_BUFFER BUF [0] flag/size/offset/xbuf_file 0x1 26 0 ./xxx
```

To read buffer data into a file (for example, *./yyy* from an **/account** object **PIN_FLD_INTERNAL_NOTES** field), do the following. The flist is stored in the **rd.flist** file:

```
>testnap
 ===> database 0.0.0.1 from pin.conf "userid"
> r rd.flist 1
> d 1
0   PIN_FLD_POID POID [0] $DB /account 1
0   PIN_FLD_INTERNAL_NOTES BUF [0] flag/size/offset/xbuf_file 0x3 26 0 ./yyy
>r flds 1
```

The contents of the PIN_FLD_INTERNAL_NOTES field is put into the file *./yyy*.

> **Note:**
>
> You cannot specify file offsets when reading from or writing to files.

Error Buffer

The PIN_FLDT_ERRBUF data type is used to record errors by the Portal Communication Module (PCM) opcodes and Portal Information Network (PIN) library macros. You call the error- or message-logging macros in the PIN library to detect the errors and to record the details of the error in a standard format.

For information on error handling in BRM, see "Finding Errors in Your Code". For descriptions of all the macros available for logging messages and errors, see "Error-Handling Macros" in *BRM Developer's Reference*.

For a complete list of the errors and values discussed in this section, see *BRM_home***/include/ pin.errs.h**.

pin_errbuf_t has the following structure:

```
typedef struct {
             int32            location;
             int32            pin_errclass;
             int32            pin_err;
             pin_fld_num_t    field;
             int32            rec_id;
             int32            reserved;
             int32            line_no;
             char            *filename;
             int              facility;
             int              msg_id;
             int              err_time_sec;
             int              err_time_usec;
             int              version;
             pin_flist_t     *argsp;
             pin_errbuf_t    *nextp;
             int              reserved2
} pin_errbuf_t;
```

Table 7-4 contains the definitions of each field in the **pin_errbuf** structure.

Table 7-4 Field Definitions in pin_errbuf

Field	Possible Values
location	Specifies the BRM module that encountered the error. Possible values are: • PIN_ERRLOC_APP The error occurred within an application. Use this value to specify that the problem originated in your application as opposed to a part of BRM. • PIN_ERRLOC_FLIST The error occurred within an flist manipulation routine local to the application. Common causes include illegal parameters and low system memory. • PIN_ERRLOC_POID The error occurred within a POID manipulation routine local to the application. Common causes include illegal parameters and low system memory. • PIN_ERRLOC_PCM The error occurred within a PCM routine local to the application. Common causes include illegal parameters. • PIN_ERRLOC_PCP The error occurred within the internal PCP library. This library provides communication support between the modules of the BRM. Common causes include network connection failures. This value indicates a system problem that requires immediate attention. • PIN_ERRLOC_CM The error occurred within the Connection Manager. Common causes include an unknown opcode or an input flist missing the required POID field. • PIN_ERRLOC_FM The error occurred within a Facilities Module. Common causes include an input flist that does not conform to the required specification. • PIN_ERRLOC_DM The error occurred within a Data Manager. Common causes include an input flist that does not meet the required specifications or a problem communicating with the BRM database.

Table 7-4 (Cont.) Field Definitions in pin_errbuf

Field	Possible Values
pin_errclass	Describes the class of error that occurred. Error class is used by an application to determine the appropriate type of error recovery. Possible values are: • PIN_ERRCLASS_APPLICATION The error was caused by the application passing illegal data or a system failure within the client application. The error was detected before the requested operation was performed, so no data in the database has changed. After the error is fixed, you can retry the operation. • PIN_ERRCLASS_SYSTEM_RETRYABLE The error was probably caused by a transient condition. You can try the operation again. Common causes include a possibly temporary shortage of system resources or failure of a network connection that you can route around. The error was detected before any data was committed to the database; no data has changed. • PIN_ERRCLASS_SYSTEM_DETERMINATE The error was caused by a system failure during the operation. Retrying the operation is unlikely to succeed, and the system failure should be investigated immediately. The error was detected before any data was committed to the database; no data has changed. After the error is fixed, you can retry the operation. • PIN_ERRCLASS_SYSTEM_INDETERMINATE The error was caused by a system failure during the commit phase of an operation. There is a small window during the commit where a network failure can leave the system unsure of whether the commit occurred or not. This means it is up to the application to determine whether system data has been changed. This class of error is extremely rare, but you must handle it carefully to avoid corrupting the data in the database. If you determine that no changes were made, you can resolve the system failure problem and then retry the operation.
pin_err	Describes the exact error that was encountered. If an API call is successful, pin_err is set to PIN_ERR_NONE and all other fields in the **ebuf** are left undefined. If an API call results in an error, one or more of the fields are defined with error information.
field	Identifies the field number of the input parameter that caused the error.
rec_id	Specifies the element ID of an array element that caused the error.
reserved	Designates an internal system state used by Oracle Technical Support for debugging. Contains no useful information for the application developer.
line_no	Specifies the line number within the application source file where the error was detected. The logging routines print the filename and line number from the **ebuf**, which you can use to locate the exact call to the BRM API that caused the error. Contains no useful information for the application developer except when working with Oracle Technical Support
filename	Specifies the name of the application source file where the error was detected. This can be used in conjunction with the *line_no* to quickly locate the source of an error. This information is useful for application developers only when they work with Oracle Technical Support.
facility	Specifies the code of a facility associated with BRM internationalization (I18N) features. Used with the *msg_id* value to create a localized error message.
msg_id	Specifies a unique ID number for each message within the facility identified by the *facility* code. Used with the *facility* value to create a localized error message.
err_time_sec	Outputs time in seconds when the error occurred.
err_time_usec	Outputs time in microseconds when the error occurred.

Table 7-4 (Cont.) Field Definitions in pin_errbuf

Field	Possible Values
version	Designates the version of the arguments.
*pin_flist_t | *argsp*	Used as an optional arguments flist.
*pin_errbuf_t *nextp*	Used one or more optional chained errbufs.
reserved2	Reserved for internal use

8

Using BRM SDK

Learn how to use the Oracle Communications Billing and Revenue Management (BRM) SDK to customize BRM or client applications.

In addition to the BRM SDK, you can use the Developer Center application to create storable classes and test applications.

Topics in this document:

- About BRM SDK
- BRM SDK Directory Contents
- Deploying New and Customized Components
- Compiling CMs for Purify

About BRM SDK

BRM Software Development Kit (SDK) provides the APIs, libraries, and other resources you need to perform the following tasks:

- Write client applications in C, C++, Java, and Perl.
- Write and customize policy Facilities Modules (FMs) in C.
- Write custom standard FMs in C.
- Write custom Data Managers (DMs) in C.
- Use sample applications and code as examples for your own work.
- Use debug versions of libraries, FMs, the DM, and the Connection Manager (CM).
- Develop multithreaded applications for BRM.

BRM SDK includes a common core library (**libportal.so**) that combines the previously separate PCM, PCP, and PIN libraries. (The separate libraries are also included for backward compatibility.) Other libraries, including standard and policy FMs and support for C, C++, Java, and Perl, are located in the same directory.

About PCM SDK

PCM SDK contains 64-bit PCM libraries that you require to create 64-bit client applications. PCM SDK is a part of BRM SDK and is installed along with the BRM server.

BRM SDK Directory Contents

When you install BRM SDK, the following subdirectories are included. The default installation directory for BRM SDK is *BRM_home*/**PortalDevKit**, where *BRM_home* is the directory in which the BRM server software is installed. To make it easier to find files, the BRM SDK directory structure is similar to the directory structure on BRM servers as shown in Table 8-1.

Table 8-1 SDK Directory Structure

Directory	Contents
lib	Core and FM libraries, including **libportal.so** on Linux.
bin	CM and DM executables, **testnap**, **perl.exe**.
include	Base BRM header files, including **pin_os_dynload.h**.
sys	Server system files.
sys/lib	CM source library files.
source	Top-level directory for source code.
source/sys	Policy FM source code.
source/sys/cm	CM source file and Makefile for use in building Purify versions of the CM.
source/templates	FM and DM templates.
source/samples	Top-level directory for sample code and applications.
source/samples/context	Context management code samples in C, C++, Java, and Perl.
source/samples/callopcode	Opcode-related code samples in C, C++, Java, and Perl.
source/samples/flists	Flist-related code samples in C, C++, Java, and Perl.
source/samples/apps	Sample applications in C, C++, and Java.
source/samples/apps/c/mta_samples	Sample files for creating multithreaded applications.
jars	**pcm.jar, pcmext.jar**, and other Java PCM files.

Deploying New and Customized Components

After you successfully test an application, FM, or DM, you can deploy it to your production BRM installation.

Because BRM SDK includes the same libraries as BRM itself, dynamic links work without modification when you deploy new or customized server components to default locations.

Deploying Applications

To deploy a new application, you move the executable itself plus any necessary support libraries to the desired location. The libraries you must include depend on the language you used to write the application. For example, applications written in C need the **libportal.so** file, while applications written in Java need **pcm.jar** and **pcmext.jar** files. Depending on how your application is written, you might also need to include a configuration file (**pin.conf** for C/C++ applications or **Infranet.properties** for Java applications) for storing login information.

For more information about the files required, see the sections about writing client applications in the supported languages.

In most cases, you should package your application and its support files so that the files can be installed conveniently.

Deploying FMs

To deploy a new or customized FM:

1. Compile the FM into a shared library (**.so** for Linux).
2. For each CM server in the BRM installation:
   a. Stop the CM.
   b. Move the new shared library to *BRM_home***/lib**.
   c. If this is a new FM, modify the CM **pin.conf** file to include the FM.
   d. Restart the CM.

Deploying DMs

The files you must deploy with a DM depend on whether the destination server is already in use as a DM server.

- If you are deploying a new or customized DM to an existing DM server, you must move only the compiled **.so** file and the associated **pin.conf** file.
- If you are deploying to a server that has not previously been used for DMs, you need the compiled **.so**, **.a**, or **.so** file, the DM **pin.conf** file, all the libraries linked to the DM, and a **dm.exe** file.

In either case, you must modify the **pin.conf** file of all CMs that will use this DM. See "Configuring Your CM to Use the Custom DM".

You should also modify the BRM start and stop scripts to include the new DM. See "Starting and Stopping Your Custom DM".

Compiling CMs for Purify

To enable customers to build versions of the CM for use with Rational Purify, BRM SDK includes a C++ source file along with related library and include files. To build a Purify version, you modify the source file to include your custom FMs and then compile.

Table 8-2 lists the CM build files for use with Purify:

Table 8-2 CM Build Files for Purify

File	Location
cm.cpp **makefile**	*BRM_SDK_home***/source/sys/cm**
libcm_main.so (Linux)	*BRM_SDK_home***/sys/lib** Also in: *BRM_home***/sys/lib**
pin_os_dynload.h	*BRM_SDK_home***/include**

When you compile a CM, you must specify options that point to the **/include** and **/lib** directories that contain the **pin_os_dynload** and **libcm_main** files. You must also specify the use of multithreaded components and dynamically loaded libraries. (The **makefile** in *BRM_SDK_home***/source/sys/cm** includes these options.)

To reduce thread contention on **malloc** calls, CMs include a memory pool mechanism for processing flists and POIDs. When flists and POIDs are allocated memory from a pool, problems with memory leaks are hidden. To detect memory leaks in your CM, before you run Purify or any other diagnostic utilities to test memory usage, disable the memory pool by

adding the following entry in the CM **pin.conf** file so that memory is allocated from the system heap:

```
- - disable_pcm_mempool 1
```

You can use the following commands as examples for compiling a Purify version of the CM. Depending on your operating system and the compiler you use, your syntax may be somewhat different.

Linux:

```
purify gcc -o cm cm.cpp -g -Bdymanic -Wl,--export-dynamic -ldl -lpthread -I $
{INFRANET_SDK_home} -L ${INFRANET_SDK_home}-lpinsys -lcm_main
```

9

Finding Errors in Your Code

Learn about the Oracle Communications Billing and Revenue Management (BRM) error logging and handling routines and how to use them in your custom applications.

The BRM APIs include a set of routines to handle and log errors. BRM uses these routines for internal error handling. You must use these routines in your custom applications to allow seamless detection and reporting of errors between your applications and the BRM applications.

Topics in this document:

- Detecting Errors in Your Code
- Error Handling Flow
- Logging Errors and Messages
- Diagnosing Application Problems
- Detecting CM and DM Errors

For information on the log file locations, syntax of error messages, and descriptions of error codes, see "Reference Guide to BRM Error Codes" in *BRM System Administrator's Guide*.

For additional information on error handling in C++, see "Handling Exceptions".

Detecting Errors in Your Code

The error buffer, **pin_errbuf_t**, is the basic structure for receiving error status from calls to the BRM Application Programming Interface (API). A pointer to an error buffer, **ebufp**, is passed into each API call and is filled in by the routine with information about any error condition that occurred.

For details on the structure and fields in an **ebufp**, see "Error Buffer".

You use the PIN_ERRBUF_IS_ERR macro in your code to test the **ebufp** for an error condition. This macro returns zero if no error exists in the **ebufp** and nonzero if an error is recorded.

All higher-level BRM API routines use the **ebufp** for error detection. These routines check for errors in the following ways:

- Check for errors after each API call. See "Individual-Style ebuf".
- Check for errors at the end of a series of API calls. See "Series-Style ebuf".

Individual-Style ebuf

PCM_*() routines log error information in the **ebufp** after each API call. Since the Portal Communication Module (PCM) API routines affect data, you must detect the errors immediately and test the status of the **ebufp** after each call to a **PCM_*()** routine. If you do not detect errors after each call, any error recorded in the **ebufp** will be overwritten by another API call, and you will lose information about the errors.

For sample code on checking the **ebufp** for errors after each **PCM_*()** call, see **sample_app.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Series-Style ebuf

PIN_*() routines *update* the error status in the **ebufp** after each API call. With the series **ebufp** style, you can perform a series of related API calls, such as creating and populating an flist, and check for errors at the end of the series. The first error is recorded in the **ebufp** and all subsequent calls are treated as no-ops so that the first error remains recorded in the **ebufp**. When you check for errors after a series of API calls, you can fix the errors that have been detected as necessary.

Using series **ebufp** style makes manipulating flists and Portal object IDs (POIDs) much more efficient, since the entire logical operation can be completed, then tested once for any errors.

> **Note:**
>
> You can check for errors any time using series-style error detection, but series-style **ebuf** has been designed to reduce the number of error checks required.

For sample code on checking the **ebufp** for errors after a series of **PIN_*()** routines, see **sample_app.c**.

Error Handling Flow

Applications that call BRM API routines must follow this general flow for error handling:

1. Declare an error buffer.
2. Call PIN_ERRBUF_CLEAR to initialize the error buffer.
3. Call PIN Library routines to create an input flist.
4. Check for errors by calling the PIN_ERRBUF_IS_ERR.
5. Call a PCM API routine.
6. Check for errors by calling PIN_ERRBUF_IS_ERR().
7. Call PIN_ERRBUF_RESET to reset the **errbuf** and cleans up the memory pointed to by the **argsp** and **nextp** pointers.

> **Note:**
>
> Previous versions of BRM used the PIN_ERR_CLEAR_ERR macro to initialize the error buffer and to reset its contents to 0. This macro still appears in some BRM code to support backward compatibility. In new code that you write, use PIN_ERRBUF_CLEAR and PIN_ERRBUF_RESET to initialize and reset the error buffer.

The following example shows this flow. For a complete sample, see **sample_app.c**.

```
...
/** Declare error buffer */
pin_errbuf_tebuf;
```

```
/** Clear the error buffer */
PIN_ERRBUF_RESET(&ebuf);

/** PIN Library routines.
 */
vp = (void *)"test";
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_BILL_MODE, vp, &ebuf);
/* Force CURRENCY to DOLLARS */
dummy = PIN_CURRENCY_DOLLARS;
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_CURRENCY, (void *)&dummy, &ebuf);
/* Set BILL_TYPE to internal (no charges). */
btype = PIN_BILL_TYPE_INTERNAL;
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_BILL_TYPE, (void*)&btype, &ebuf);
/* Add a nameaddr array element to the flist. */
a_flistp = PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_NAMEINFO,
PIN_NAMEINFO_BILLING, &ebuf);
vp = (void *)"Doe";
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_LAST_NAME, vp, &ebuf);
vp = (void *)"John";
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_FIRST_NAME, vp, &ebuf);
vp = (void *)"1234 Main Street";
PIN_FLIST_FLD_SET(a_flistp, PIN_FLD_ADDRESS, vp, &ebuf);

 /* Add more PIN library routines to gather all the information needed to register
    a customer */

/* Check for errors */
if (PIN_ERRBUF_IS_ERR(&ebuf)) {
PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_ERROR,
"sample_init_input error", &ebuf);
PIN_ERRBUF_RESET(&ebuf);
return;
    }
/** Clear the error buffer */
PIN_ERRBUF_RESET(&ebuf);


/* PCM operations */
/* Call the COMMIT CUSTOMER opcode */

 PCM_OP(ctxp, PCM_OP_CUST_COMMIT_CUSTOMER, 0, flistp, &r_flistp, &ebuf);

/* Check for errors */
if (PIN_ERRBUF_IS_ERR(&ebuf)) {
PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_ERROR,
 "create_customer error"", &ebuf);
PIN_ERRBUF_RESET(&ebuf);

return;
    }
```

Logging Errors and Messages

When detecting errors using PIN_ERRBUF_IS_ERR, you can call the error- or message-logging macros to record the details of the error in a standard format. For example, you can use the PIN_ERR_LOG_EBUF macro to print the contents of an **ebufp** along with a custom message to your application's logfile.

To log any messages, including errors unrelated to ebufs and the BRM API, use the PIN_ERR_LOG_MSG macro at any point in your application.

Table 9-1 lists the routines you can use in your application to log status information:

Table 9-1 Routines Used to Log Status Information

Action	Routine
Specify the path and name of the log file for your application. The default log file is **default_pin.log** in the application directory.	PIN_ERR_SET_LOGFILE
Specify your application's name in the log entries to identify the program in which the error occurred.	PIN_ERR_SET_PROGRAM
Specify what types of messages to log and what to discard. You can enable debugging messages during development and then turn them off by changing the log level setting.	PIN_ERR_SET_LEVEL
Print the contents of an flist to the error log.	PIN_ERR_LOG_FLIST
Print the contents of a POID to the error log.	PIN_ERR_LOG_POID

For a list and description of all the macros available for logging messages and errors, see "Error-Handling Macros" in *BRM Developer's Reference*.

For an explanation of the standard log entry format, see "Reference Guide to BRM Error Codes" in *BRM System Administrator's Guide*.

Diagnosing Application Problems

To diagnose application problems:

1. Check the information in the **ebuf** for the exact source location of the API call that generated the error. If the error is caused by an incorrect or missing field on the input flist, the **ebuf** provides the field name.

   For details about the error buffer format and contents, see "Error Buffer".

2. If you cannot diagnose an error with the information in the **ebuf**, use PIN_ERR_SET_LEVEL to enable the application libraries to log debug messages. Errors that occur in the application libraries are printed in detail to the logfile. This helps you locate errors like illegal **NULL** pointers.

Detecting CM and DM Errors

Enable the CMs and DMs to log debug messages to be printed when an operation fails because of bad input. These messages are printed to the CM and DM log files, not to the application's log file. Normally this type of error is not logged because it is not caused by a failure in the BRM system. You can enable and disable debug messages by editing the CM and DM configuration files.

For information on enabling error logging in the CM and DM configuration files, see the configuration files in the CM and DM directories.

10

Testing Custom Applications

Learn how to test custom applications in Oracle Communications Billing and Revenue Management (BRM).

Topics in this document:

- Testing New or Customized Components
- Changing the Virtual System Time to Test BRM

Testing New or Customized Components

The BRM architecture makes it possible to test new or customized components without having to physically move files. For example, to test a new policy FM, you can include it in a CM that you run locally on your development machine.

For testing, you need access to a test installation of BRM. For best results, the test installation should resemble very closely your production BRM environment.

The following sections provide basic instructions for common testing scenarios. Depending on the nature of your customizations and the architecture of your BRM system, these instructions might not fully describe your situation.

Testing Custom Applications

To test an application, connect it to a BRM system via its CM, in the same way you would under production conditions. To establish a connection, the application must specify a valid user name, password, port number, and database number. Depending on how the application is designed, this information can be included in a configuration file (**pin.conf** for C/C++ applications or **Infranet.properties** for Java applications) or specified by the user.

Testing New or Customized Policy FMs

To test a new or customized policy FM, you must add it to a CM that you then use with a BRM test installation.

1. Add the FM to the CM in BRM SDK. See "Adding a New FM Module to the CM Configuration File".
2. Use that CM in place of the default CM in a test installation of BRM. See "Configuring Your CM to Use the Custom DM".
3. Use Opcode Workbench or **testnap** to run the opcodes in the new or customized FM. See "Using the testnap Utility to Test BRM".

   Keep in mind that if the opcodes in the FM require the use of new storable classes or fields, you must add those storable classes or fields to the database of the test BRM installation. See "Creating Custom Fields and Storable Classes".

See "Debugging FMs" for information about debugging new or customized policy FMs.

Testing New or Customized DMs

To test a new or customized DM, run a CM locally on your development machine. The CM should connect to the DM that you are testing and to any other DMs included in the BRM installation. For information about including the new or customized DM in the CM, see "Configuring Your CM to Use the Custom DM".

Changing the Virtual System Time to Test BRM

You can test some BRM functionality by using the **pin_virtual_time** utility. For example, you can test the billing impact of recurring charges by advancing the date within the BRM system and then running billing.

> **Caution:**
>
> Always use a test database when you test BRM functionality.

To test some aspects of billing, you must simulate the passage of time. The **pin_virtual_time** utility enables you to simulate changes in the BRM system time.

> **Note:**
>
> The **pin_virtual_time** utility works only on a single computer. Typically, you set up a test BRM system on just one computer. If you run **pin_virtual_time** on a system that is distributed across multiple computers, you must carefully coordinate time changes on all the computers.

To set up the pin_virtual_time utility:

1. Go to the *BRM_home***/sys/test** directory.
2. Run this command to create a file that **pin_virtual_time** requires, called **pin_virtual_time_file**:

   ```
   pin_virtual_time -m 0 -f BRM_home/bin/pin_virtual_time_file
   ```

   > **Note:**
   >
   > *BRM_home***/lib/pin_virtual_time_file** is the standard path and file name, but you can change it.

3. Add the following entry to the configuration file in the *BRM_home***/sys/test** directory:

   ```
   - - pin_virtual_time  pin_virtual_time_file
   ```

   Replace *pin_virtual_time_file* with the path and name of the mapped file created in the previous step.

Adding the entry to the configuration file in **sys/test** enables you to run **pin_virtual_time** from that directory. The directory from which you run **pin_virtual_time** must contain a configuration file with the **pin_virtual_time** entry.

4. Add the entry in step 3 to the configuration file for each program you want to respond to an altered time.

   To test your price list, edit the configuration files for at least the applications listed in Table 10-1:

Table 10-1 Configuration Files for Testing Applications

Application	Configuration File Location
CM	**sys/cm/**
Oracle DM	**sys/dm_oracle/**
Billing and invoice utilities	**apps/pin_billd/**

   To ensure that all applications respond to **pin_virtual_time** changes, add the entry to all the configuration files in your test BRM system.

   Entry example:

   ```
   - - pin_virtual_time  BRM_home/bin/pin_virtual_time_file
   ```

After you set up the **pin_virtual_time** utility, run **pin_virtual_time** to advance the BRM date and time.

To run **pin_virtual_time**, use the following syntax:

```
pin_virtual_time -m 2 MMDDHHMM[CC]YY.[SS]
```

For the string MMDDHHMM[CC]YY.[SS], enter the date and time you want BRM to use in this format: month, date, hour, minute, year, and seconds. You must enter at least two digits for the year, but you can enter four. Seconds are optional.

For example, to set the date and time to 9/3/99 and 11:30, respectively:

```
% pin_virtual_time -m 2 090311301999.00
```

The command displays this message at the command prompt:

```
filename BRM_home/lib/pin_virtual_time_file, mode 2, time: Fri Sept 03 11:30:00 1999
```

The time then advances normally from the new reset time.

After you run **pin_virtual_time**, you must stop and restart BRM to read in the new time.

11

Using the testnap Utility to Test BRM

Learn how to use the **testnap** utility to test your Oracle Communications Billing and Revenue Management (BRM) applications and custom code.

> **Note:**
>
> Only the most common **testnap** functions are covered here. For a complete description, see "testnap".

Topics in this document:

- About testnap
- Executing Opcodes
- Reading an Object and Fields
- Retrieving Objects
- Creating Objects
- Manipulating External Buffer Fields
- Sorting an Flist
- Invoking Shell Commands
- Troubleshooting testnap

See also:

- Using testnap to Modify /config Objects
- Using BRM SDK
- Finding Errors in Your Code
- Testing Custom Applications

About testnap

The **testnap** utility enables a developer to manually interact with the BRM server by establishing a PCM connection with the Connection Manager (CM) and then executing PCM operations by using that connection. Using **testnap**, a developer can perform the following tasks:

- Create input flists
- Save input and output flists
- Send opcodes
- View return (output) flists
- Create, view, modify, and delete objects and their fields

- Open, commit, and cancel transactions

Opcode Workbench, part of the Developer Center offers similar functionality in a GUI-based application.

About Buffer Numbers

The **testnap** utility allocates numerous internal buffers, which are used to store object or flist fields. Buffers are referenced by integers of the user's choice. Every time a new buffer is referenced, **testnap** allocates that new buffer.

- If you do not specify a buffer number for a command that expects one, you will be prompted for a buffer number.
- The **meta** keyword causes **testnap** to display the size of external buffer fields. By default, the contents of external buffer fields are displayed.

Executing Opcodes

To run opcodes using **testnap**, use the **xop** command.

```
xop opcode number flag buffer number
```

The input flist is read from a file into **buffer 1**, which is passed to **xop**, on the **xop** command line. See "About Buffer Numbers".

For opcode numbers, see the opcode header files in *BRM_home***/include/ops**.

Reading an Object and Fields

You can use **testnap** to read objects or fields in an object and write its contents to a file.

See the following:

- Reading Fields in an Object
- Reading an Object and Writing Its Contents to a File

Reading Fields in an Object

To read fields from an object, each field or row in the field list must be in valid flist format, but the actual values for the last two fields need not be valid.

For example, you have to include " " for STR fields and some number for the TSTAMP field. If a field is blank, the system returns an error. Be sure to include the header line in all flists.

1. Print the flist to ensure that the format is correct and all the fields are filled.
2. Start **testnap**.

   ```
   testnap
   ```

3. Read the fields.

   ```
   r fldlist 1
   rflds 1
   ```

Reading an Object and Writing Its Contents to a File

This example reads the contents of the **/account** object and writes it in to a file called **root.account**.

1. Start **testnap**.

   ```
   testnap
   ===> database 0.0.0.1 from pin.conf "userid"
   ```

2. Read the object you want.

   This example reads the **/account** object.

   ```
   testnap

   robj - $DB /account 1
   ```

3. Save output in buffer 1.

   ```
   s 1
   ```

4. Write the contents on buffer 1 into a file, and quit.

   This example writes the contents into a file called **root.account**.

   ```
   w 1 root.account
   q
   ```

5. Print the contents of the file (**root.account** in the example) to verify that the file contains the contents of the **/account** object.

   ```
   cat root.account
   ```

Retrieving Objects

You can use **testnap** to search for objects and retrieve the contents of the objects, of specific objects, or the Portal object IDs (POIDs) of the objects.

See the following:

- Retrieving the Contents of the First Object Found
- Retrieving the POID Field of the Objects Found
- Creating a New Search Object
- Retrieving Objects One at a Time
- Retrieving a Specific Number of Objects at a Time

Retrieving the Contents of the First Object Found

This example shows how to perform a search and retrieve the contents of the first object found by that search.

```
cat search

0 PIN_FLD_POID                    POID [0] 0.0.0.1 /search 201
0 PIN_FLD_ARGS                   ARRAY [1]
1     PIN_FLD_POID                POID [0] 0.0.0.1 /account 1 1
0 PIN_FLD_RESULTS                ARRAY [0]
```

```
testnap

r search 1
```

Retrieving the POID Field of the Objects Found

This example shows how to search for objects and retrieve only the POID field of each object found.

```
cat search.all_acct

0 PIN_FLD_POID                   POID [0] 0.0.0.1 /search 201
0 PIN_FLD_ARGS                  ARRAY [1]
1     PIN_FLD_POID               POID [0] 0.0.0.1 /account -1 1
0 PIN_FLD_RESULTS               ARRAY [0]
1     PIN_FLD_POID               POID [0] 0.0.0.1 /account 0 0

testnap

r search.all_acct 2
```

Creating a New Search Object

You can create a new search object, load it in the database, and use it as a template for your searches.

The following example shows you how to create a new search template.This template can be used in programs to search for search objects in the database.

1. Read an existing search object and save it as a template:

```
testnap

robj - $DB /search 222

# number of field entries allocated 6, used 6
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search 222 1
0 PIN_FLD_NAME               STR [0] "1 result: sum, 6 arg =, >=, <, =, like, >
special search in /event/usage"
0 PIN_FLD_CREATED_T       TSTAMP [0] (857152638) Fri Feb 28 09:57:18 1997
0 PIN_FLD_MOD_T           TSTAMP [0] (857152638) Fri Feb 28 09:57:18 1997
0 PIN_FLD_FLAGS              INT [0] 1
0 PIN_FLD_TEMPLATE           STR [0] "select sum( F1 ) from /event/usage where
                                      event_total_t.rec_id = 1
                                      and F2 = V2 and F3 >= V3 and F4 < V4
                                      and F5 = V5 and F6 like V6 and F7 > V7"

s 1

saved input in buffer 1

w 1 search.template
q
```

2. Edit the template file to suit your needs.
3. Start **testnap**.
4. Create the new search template, that is, a new search object with the number **977**, as shown in the following example:

```
cat search.template
```

```
    0 PIN_FLD_POID           POID [0] 0.0.0.1 /search 977
    0 PIN_FLD_NAME            STR [0] "1 arg = search for /search/$1"
    0 PIN_FLD_FLAGS           INT [0] 1
    0 PIN_FLD_TEMPLATE        STR [0] "select X from /search/$1 where F1 = V1 "

    testnap

    # Read the search template in to buffer 1

    r search.template 1

    # Create an object using the contents of buffer 1

    create 1 poid
       poid created was: 0.0.0.1 /search 977 0

    # Verify that the search object was created

    robj - $DB /search 977

    # number of field entries allocated 6, used 6
    0 PIN_FLD_POID          POID [0] 0.0.0.1 /search 977 0
    0 PIN_FLD_NAME           STR [0] "1 arg = search for /search/$1"
    0 PIN_FLD_CREATED_T   TSTAMP [0] (857521039) Tue Mar  4 16:17:19 1997
    0 PIN_FLD_MOD_T       TSTAMP [0] (857521039) Tue Mar  4 16:17:19 1997
    0 PIN_FLD_FLAGS          INT [0] 1
    0 PIN_FLD_TEMPLATE       STR [0] "select X from /search/$1 where F1 = V1 "
```

testnap displays **/search** object 977 from the database.

Retrieving Objects One at a Time

You can use **testnap** to do a step search. This example shows how to retrieve objects found in a step search one at a time:

```
t_flist2

0 PIN_FLD_POID          POID [0] 0.0.0.1 /search 236 0
0 PIN_FLD_ARGS         ARRAY [1]
1       PIN_FLD_PASSWD   STR [0] "ozt|%"
0 PIN_FLD_RESULTS      ARRAY [1]
1       PIN_FLD_POID    POID [0] 0.0.0.0 / 0 0
1       PIN_FLD_LOGIN    STR [0] ""
1       PIN_FLD_PASSWD   STR [0] ""

testnap

r t_flist2 1
ssrch 1

# number of field entries allocated 2, used 2
0 PIN_FLD_POID             POID [0] 0.0.0.1 /search 236 0
0 PIN_FLD_RESULTS         ARRAY [0] allocated 3, used 3
1     PIN_FLD_POID         POID [0] 0.0.0.1 /service/pcm_client 1 1
1     PIN_FLD_LOGIN         STR [0] "root.0.0.0.1"
1     PIN_FLD_PASSWD        STR [0] "ozt|5f4dcc3b5aa765d61d8327deb882cf99"

snext

buffer: 1
# number of field entries allocated 2, used 2
0 PIN_FLD_POID             POID [0] 0.0.0.1 /search 236 0
```

```
0 PIN_FLD_RESULTS         ARRAY [0] allocated 3, used 3
1     PIN_FLD_POID         POID [0] 0.0.0.1 /service/admin_client 2 1
1     PIN_FLD_LOGIN         STR [0] "root.0.0.0.1"
1     PIN_FLD_PASSWD     STR [0] "ozt|5f4dcc3b5aa765d61d8327deb882cf99"

send

buffer: 1
# number of field entries allocated 3, used 3
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search 236 0
0 PIN_FLD_ARGS          ARRAY [1] allocated 1, used 1
1     PIN_FLD_PASSWD      STR [0] "ozt|%"
0 PIN_FLD_RESULTS       ARRAY [1] allocated 3, used 3
1     PIN_FLD_POID       POID [0] 0.0.0.0 / 0 0
1     PIN_FLD_LOGIN       STR [0] ""
1     PIN_FLD_PASSWD      STR [0] ""
```

Retrieving a Specific Number of Objects at a Time

You can change the number of elements in the PIN_FLD_RESULTS ARRAY specification to specify the number of objects you want to retrieve at a time.

This example shows retrieving three objects each time:

```
cat t_flist_arr

0 PIN_FLD_POID          POID [0] 0.0.0.1 /search 236 0
0 PIN_FLD_ARGS         ARRAY [1]
1       PIN_FLD_PASSWD   STR [0] "ozt|%"
0 PIN_FLD_RESULTS      ARRAY [3]
1       PIN_FLD_POID    POID [0] 0.0.0.0 / 0 0
1       PIN_FLD_LOGIN    STR [0] ""
1       PIN_FLD_PASSWD   STR [0] ""

testnap
===> database 0.0.0.1 from pin.conf "userid"

r t_flist_arr 1
ssrch 1

# number of field entries allocated 4, used 4
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search 236 0
0 PIN_FLD_RESULTS          ARRAY [0] allocated 3, used 3
1     PIN_FLD_POID          POID [0] 0.0.0.1 /service/pcm_client 1 1
1     PIN_FLD_LOGIN          STR [0] "root.0.0.0.1"
1     PIN_FLD_PASSWD         STR [0] "ozt|5f4dcc3b5aa765d61d8327deb882cf99"
0 PIN_FLD_RESULTS          ARRAY [1] allocated 3, used 3
1     PIN_FLD_POID          POID [0] 0.0.0.1 /service/admin_client 2 1
1     PIN_FLD_LOGIN          STR [0] "root.0.0.0.1"
1     PIN_FLD_PASSWD         STR [0] "ozt|5f4dcc3b5aa765d61d8327deb882cf99"
0 PIN_FLD_RESULTS          ARRAY [2] allocated 3, used 3
1     PIN_FLD_POID          POID [0] 0.0.0.1 /service/email 1061 3
1     PIN_FLD_LOGIN          STR [0] "kim@csi.com"
1     PIN_FLD_PASSWD         STR [0] "ozt|ae2b1fca515949e5d54fb22b8ed95575"

q
```

Creating Objects

To create an object, you must create an flist in a text file or as a **here** document in a **testnap** script. The following procedures show two ways to create an object using **testnap.**

See the following:

- Using a Text File to Create an Object
- Using a Here Document to Create an Object

Using a Text File to Create an Object

To use a text file to create an object:

1. Create an flist using a text editor with at least the required fields of the object by using the following example:

```
0 PIN_FLD_POID              POID [0] 0.0.0.1 /data 0 0
0 PIN_FLD_NAME              STR [0] "example new data object creation"
0 PIN_FLD_BUFFER            BUF [0] flag/size/offset 0x0 0 0 NULL data ptr
0 PIN_FLD_HEADER_NUM        INT [0] 1234
0 PIN_FLD_HEADER_STR        STR [0] "Some sample info in the example"
0 PIN_FLD_PARENT            POID [0] 0.0.0.0  0 0
```

2. Save the file.

   In this example, the file is called **new_data_flist**

3. Start **testnap**.

```
pin@demo5-661> testnap
===> database 0.0.0.1 from pin.conf "userid"
```

4. Read the file into a buffer.

```
r new_data_flist 1
```

5. Create an object using that buffer.

```
create 1
poid created was: 0.0.0.1 /data 8830 0
```

   The POID of the object created is returned.

Using a Here Document to Create an Object

This procedure enables you to combine data and commands in one file:

1. Create a **testnap** script to read an object from a **here** document:

```
#
# A testnap script to create a new data object
#
# Read an flist into buffer 1 using a "here" document
# Note the space between "<<" and the "here" token.
# Note use of $DB_NO in poid database - takes current database number.
r << XXX 1
0 PIN_FLD_POID           POID [0] $DB_NO /data 0 0
0 PIN_FLD_NAME            STR [0] "example new data object creation"
0 PIN_FLD_BUFFER          BUF [0] flag/size/offset 0x0 0 0 NULL data ptr
0 PIN_FLD_HEADER_NUM      INT [0] 1234
```

```
0 PIN_FLD_HEADER_STR      STR [0] "Some sample info in the example"
0 PIN_FLD_PARENT         POID [0] 0.0.0.0  0 0     XXX
#
# Create the object
#
  create 1 poid
#
# Write the new poid id (from the "in" buffer)
# into a file "new_data_poid.<pid_of_this_process>"
#
w in new_data_poid.$$
#
```

2. Save the script.

   In this example the script is saved as **new_data_script**.

3. Run the script.

   You can either run **testnap** with the script name as the argument, as shown in this example, or use the **<** command in **testnap**.

   See **testnap** for details.

   ```
   ./testnap new_data_script

   poid created was: 0.0.0.1 /data 9854 0

   ls new_data_poid*

   new_data_poid.3881
   ```

4. Display the contents of the file to verify that the output file is created.

   ```
   cat new_data_poid.3881

   # number of field entries allocated 1, used 1
   0 PIN_FLD_POID        POID [0] 0.0.0.1 /data 9854 0
   ```

Manipulating External Buffer Fields

You can use **testnap** to manipulate the external buffer fields, for example, to read data in a buffer to a file.

See the following:

- Reading Data in a Buffer to a File
- Using Buffers to Concatenate Flists
- Setting up Buffers and Displaying the List of Buffers
- Creating and Displaying the Contents of a Buffer

Reading Data in a Buffer to a File

The procedure in this section shows how to set the root account object's (**/account 1**) PIN_FLD_INTERNAL_NOTES field, which is an external buffer field in the **/account** storable class.

In the input flist in the example, the 0x1 flag indicates that the contents of the file are to be written to the object. The size parameter (37), which is required, is the number of bytes in the file. The contents of this file is to be read into the external buffer field.

The offset field, if nonzero, indicates the number of bytes to be skipped in the file before reading *size* bytes from the file and placing them in the INTERNAL_NOTES field.

1. Set up an input flist that specifies that the contents of the file **xbuf.out** be placed into the INTERNAL_NOTES field of the root account object when the Write Fields opcode is run.
2. Setup an input flist that will read the INTERNAL_NOTES field of the root account object when the Read Fields opcode is run. The 0x3 flag indicates that the contents of the file are to be read from the object.
3. Perform the Read Fields opcode. The contents of the INTERNAL_NOTES field are written to the newly created file **xbuf.in**.

   Example:

```
file xbuf.out

this is test of xbuf to account 1
0 PIN_FLD_POID           POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_INTERNAL_NOTES  BUF [0] flag/size/offset/xbuf_file 0x1 37 0 xbuf.out

xbuf.write

# number of field entries allocated 45, used 45
0 PIN_FLD_POID           POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_INTERNAL_NOTES  BUF [0] flag/size/offset/xbuf_file 0x1 37 0 xbuf.out

buf.tst
# number of field entries allocated 45, used 45
0 PIN_FLD_POID           POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_INTERNAL_NOTES  BUF [0] flag/size/offset/xbuf_file 0x3 0 0 xbuf.in

testnap
r xbuf.write 1
wflds 1
r xbuf.tst 2
rflds 2
q
```

Using Buffers to Concatenate Flists

You can use **testnap** to concatenate flists. For example:

1. Read the contents of two different files into two different buffers (buffer 1 and buffer 2).
2. Write the contents of buffer 1 to a file.
3. Append the contents of buffer 2 to the same file.

   Example:

```
cat bill

0 PIN_FLD_POID POID [0] 0.0.0.1 /bill 1451 0

cat bundle
0 PIN_FLD_POID                    POID [0] 0.0.0.1 /deal 1123 0

testnap

r bill 1
r bundle 2
w 1 billbundle
w+ 2 billbundle
```

```
q
cat billbundle

# number of field entries allocated 20, used 1
0 PIN_FLD_POID                    POID [0] 0.0.0.1 /bill 1451 0
# number of field entries allocated 20, used 1
0 PIN_FLD_POID                    POID [0] 0.0.0.1 /deal 1123 0
```

Setting up Buffers and Displaying the List of Buffers

You can use **testnap** to display a list of all the buffers on your system.

In this example, the flists in two different files (**bill** and **bundle**) are read into two different buffers. Then the list of all objects in all the buffers are displayed:

```
cat bill

0 PIN_FLD_POID               POID [0] 0.0.0.1 /bill 1451 0

cat bundle

0 PIN_FLD_POID               POID [0] 0.0.0.1 /deal 1123 0
pin@demo5-511> testnap
===> database 0.0.0.1 from pin.conf "userid"

r bill 1
r bundle 2
l

[1] type /bill, poid 1451
[2] type /deal, poid 1123
```

Creating and Displaying the Contents of a Buffer

To create and display the contents of a buffer:

1. Read the contents of the file called **bill** into **buffer 1**.

   ```
   r bill 1
   ```

2. Display the contents of **buffer 1**.

   ```
   d 1

   # number of field entries allocated 20, used 1
   0 PIN_FLD_POID           POID [0] 0.0.0.1 /bill 1451 0
   ```

Sorting an Flist

On Linux, by using the **sort** option in **testnap**, you can sort the contents of a buffer.

This example shows how to read an flist into the buffer and sort it:

```
cat products

cat products.sort

0 PIN_FLD_PRODUCTS                  ARRAY [0] allocated 20, used 1
1     PIN_FLD_QUANTITY                NUM [0] 1.000000

testnap
```

```
===> database 0.0.0.1 from pin.conf "userid"

r products 2
r products.sort 1
sort 2 1
```

Invoking Shell Commands

On Linux, you can invoke shell commands from **testnap**.

This procedure shows how the results of a **grep** invocation are used to determine what exit code to use.

1. Search the database to see if **/service/ip testterm01** is already created.

```
r << XXX 1

0 PIN_FLD_POID           POID [0] $DB_NO /search 236 0
0 PIN_FLD_PARAMETERS      STR [0] "ip"
0 PIN_FLD_ARGS          ARRAY [1]
1     PIN_FLD_LOGIN       STR [0] "testterm01"
0 PIN_FLD_RESULTS       ARRAY [0]
1     PIN_FLD_POID       POID [0] 0.0.0.0  0 0
1     PIN_FLD_LOGIN       STR [0] ""
XXX

search 1
```

2. Write the results to a file.

```
w in out.setup.fm_term.$$/exist.testterm01
```

> **Note:**
>
> **$$** is substituted for the current process id on Linux, file names in the **r**, **r+**, **w**, **w+,** and **<** commands, and in the arguments to the **!** command.

3. Use a Linux shell to perform an if-test on the results of a **grep** invocation, and then set the exit code accordingly.

```
! if grep "testterm01" out.setup.fm_term.$$/exist.testterm01 ; then exit 1; else
exit 0 ; fi
```

Troubleshooting testnap

When **testnap** doesn't run successfully, you see an error message or the **nap** prompt doesn't appear. Also, you see error messages in the **testnap** log file located by default in the *BRM_home***/sys/test** directory. The error messages include an error number and the location of where the error occurred.

Check the **testnap** log file and the **cm.log**, **cm.pinlog, dm_oracle.log**, and **dm_oracle.pinlog** files for details.

Most problems with starting or running **testnap** involve incorrect parameters in the **testnap** configuration file.

If **testnap** doesn't start, you see the following message:

```
E Thu May B 13:16:31 1999  db2.corp  :6029 pcm.c(1.40):90
     Connect open failed (4/100) in pcm context open
E Thu May B 13:22:07 1999  db2.corp  :6033  pcm.c(1.40):90
     Connect open failed (4/101) in pcm context open
```

See the following:

- Error 27: Connection Error
- Error 4: Login Failure
- Incorrect Database Number
- Error 26: DM Not Running
- Invalid Buffer Index
- Error 56: Failed to Connect

Error 27: Connection Error

This error is caused by one of the following situations:

- The maximum number of connections have been exceeded. This is indicated by the following message:

  ```
  (10567): pcp_open, bad connect: Connection refused
  (10567): login failed 27
  ERROR: testnap: pcm_context_open():: err 27,
           loc 0, pin_errclass 0, field 0/0, rec_id 0, resvd 5
  ```

  You can resolve the problem by increasing the number of connections in the CM **pin.conf** file and reconfiguring the CM, or by connecting to a different CM.

- The DM processes did not start when you tried to start **testnap**, caused by the following sequence of commands:

  ```
  pin@demo5-86> pin_ctl start dm_oracle
  pin@demo5-87> pin_ctl start cm
  pin@demo5-88> testnap
  ERROR: testnap: pcm_connect():: err 27,
  loc 2, pin_errclass 1, field 0/0, rec_id 0, resvd 7
  ```

  Wait a few seconds for the DM processes to start after typing **pin_ctl start cm** before starting **testnap**.

Error 4: Login Failure

Error 4 in the message indicates that login failed because of an incorrect port or host number or incorrect user ID.

Incorrect port number

A message such as the following indicates that the port number specified by the **cm_ports** entry in the **testnap pin.conf** file is incorrect:

```
testnap
(8488): bad receive of login response, err 4
(8488): login failed 4
ERROR: testnap: pcm_context_open():: err 4,
         loc 0, pin_errclass 0, field 0/0, rec_id 0, resvd 5
```

To resolve the problem, make sure that the port number in the **cm_ports** entry in the **testnap pin.conf** file matches the **cm_ports** entry in the CM **pin.conf** file.

Incorrect user ID

The following message indicates that the database number (db_no) in the **userid** entry of the **testnap pin.conf** file is incorrect:

```
testnap
bad/no "userid" from pin.conf file
ERROR: testnap: pcm context open():: err 4,
     loc 0, pin errclass 0, field 0/0, rec_id 0, resvd 3
```

To resolve the problem, enter the correct database number in the **pin.conf** file.

Connection refused

The following message indicates that either the **testnap pin.conf** file has an incorrect port number or hostname in the **cm_ports** entry or there is no CM running.

```
testnap
(1215): pcp_open, bad connect: Connection refused
ERROR: testnap: pcm context open():: err 4,
      loc 0, pin_errclass 0, field 0/0, rec_id 0, resvd 4
```

To resolve the problem, make sure that the hostname and port number are correct and that the CM is running.

Incorrect hostname

The following message indicates that the hostname in the **testnap pin.conf** file is incorrect:

```
testnap
(6044): pcp_open(), bad gethostbyname("XXX_HOSTNAME"): Error 0
ERROR: testnap: pcm_context_open():: err 4,
      loc 0, pin errclass 0, field 0/0, rec_id 0, resvd 3
```

To resolve the problem, enter the correct hostname.

Incorrect Database Number

The following message indicates that **testnap** could connect to the DM through the CM, but couldn't access the database, since the database number was incorrect:

```
testnap
XXX: database 23 from pin.conf "userid"
ERROR: dd vrfy(): pcm read_flds(): 23
```

Even though the CMs and DMs successfully came up, this is the first point at which the validity of the database number is checked.

To resolve the problem, correct the database number (db_no) in the **userid** in **testnap pin.conf** file. Also, ensure that the **testnap**, CM, and DM **pin.conf** files have the same database numbers.

Error 26: DM Not Running

The following messages indicate that there is no DM running:

```
testnap
```

```
XXX: database 2 from pin.conf "userid"
ERROR: dd vrfy(): pcm read_flds(): 26

robj - 0.0.0.2 /account 1

PCM_OP_READ_OBJ failed: err 26,
          loc 3, pin_errclass I, field 0/16, rec_id 0, resvd 3
```

To resolve this, start a DM.

Invalid Buffer Index

The following message indicates that there is a parameter missing in the **testnap** command:

```
testnap

XXX: database 2 from pin.conf "userid"

robj /account 1

ERROR: invalid buffer index "2"
no object to use for robj

robj
```

To resolve the problem, be sure to include the "-" in front of the number 0.0.0.2:

```
robj - 0.0.0.2 /account 1
```

Error 56: Failed to Connect

If some older DMs started using a different database number are still running and accepting connections from your application, which is trying to use a different database, **testnap** returns error 56.

In addition, the **testnap pinlog** file contains the following message:

```
E Fri May 7 16:14:11 1999 demo5 <no name>:2025 pcm.c(1.46):101
        Connect open failed (56/7) in pcm_context_open
E Fri May 7 16:14:11 1999 demo5 <no name>:2025 pcm_conn.c(1.3):158
        pcm_connect: bad pcm_context_open, err 56
```

The CM **pinlog** file contains the following message:

```
E Fri May 7 16:35:46 1999 demo5 cm:2125 fm_utils_trans.c:114
        fm_utils_trans_open error [location=<PIN_ERRLOC_DM:4>
class=<PIN_ERRCLASS_APPLICATION:4> errno=<PIN_ERR_WRONG_DATABASE:31> field
num=<PIN_FLD_POID:7,16> recid=<0> reserved=<0>]
```

To resolve the problem, stop all the DMs and restart them.

Use **ps -ef | grep dm** to find and stop all DMs.

> **Note:**
>
> **testnap** also returns error 56 when you re-create the database but do not commit your changes in SQL or in **init_tables.sql.hostname**.

Part II
Customizing BRM Server Components

This part describes how to customize Oracle Communications Billing and Revenue Management (BRM) server components. It contains the following chapters:

- About System and Policy Opcodes
- Writing a Custom Facilities Module
- Writing a Custom Data Manager
- Creating Custom Fields and Storable Classes
- Storing Customer Profile Information
- Auditing Customer Data
- Encrypting Data
- Searching for Objects in the BRM Database
- Adding Support for a New Service
- Using BRM Messaging Services
- Using BRM with Oracle Application Integration Architecture
- Using Event Notification
- Writing Custom Batch Handlers
- Managing Devices with BRM
- Managing Orders

12

About System and Policy Opcodes

Learn how to use policy opcodes to change default Oracle Communications Billing and Revenue Management (BRM) policies and implement new business policies.

Topics in this document:

- Understanding System and Policy Facilities Modules
- Using the Policy Opcode Source Files
- Adding a New Policy FM

For a complete list of business policies implemented by default in BRM and how to change them, see "Implementation Defaults".

Understanding System and Policy Facilities Modules

BRM functionality, such as billing and rating, is implemented by using FMs and a set of applications that rely on them. The FMs implement opcodes, each of which performs a specific function related to the business processes of its manager. See "FM Opcodes" for more information.

BRM uses two types of opcodes:

- System opcodes
- Policy opcodes

Many system opcodes have a corresponding policy opcode that allows you to customize functionality. For example, the PCM_OP_AR_REVERSE_WRITEOFF standard opcode calls the PCM_OP_AR_POL_REVERSE_WRITEOFF policy opcode, which you can customize.

From a technical perspective, System FMs and Policy FMs follow the same rules and are built with the same Application Programming Interfaces (APIs). However, they differ significantly in the types of functions they implement. Each System and Policy FM set has a corresponding header file, which must be included in applications that use an opcode from that FM set.

System FM Functions

System FMs:

- Define and implement the basic functionality.
- Guarantee the integrity of all operations performed.

For example, the customer system FM includes the PCM_OP_COMMIT_CUSTOMER opcode, which takes customer account creation information and creates an account object in the database. A formal set of steps is followed, including preparing and validating all the fields, creating and initializing the account and service objects, purchasing the appropriate packages and bundles, validating the credit card, and so on. These steps are carried out within a well-defined transactional model that guarantees the integrity of the resulting objects.

You cannot modify the default System FMs, but you can create new ones and configure them into BRM.

For information on creating a custom FM, see "Writing a Custom Facilities Module".

Policy FM Functions

You use Policy FMs to implement your business policy decisions. System FMs use these policies to choose among a set of reasonable behaviors under a specific set of circumstances.

For example, the payment module includes an opcode in its Policy FM called PCM_OP_PYMT_POL_SPEC_VALIDATE, which determines whether a customer's payment information needs to be validated after events such as a new account creation or a change to the customer's payment details. This policy opcode doesn't perform any of the actual work. It takes a description of the situation, makes the decision based on your business policy, and returns that decision to the System FM for execution.

Every policy opcode includes a default implementation already configured into BRM. If your business policies differ from the default implementation, you can implement custom policies and configure them into BRM. Because the business policies are separated into their own opcodes, you can easily customize the policies without affecting the underlying functionality of BRM.

Policy FMs consist of a number of opcodes that are implemented by using C functions, and these opcodes enable you to change the default behavior of BRM. For example, you can substitute customized opcodes for existing policy opcodes to validate Automatic Account Creation (AAC) information, create and check customer passwords, validate login names, and assign dynamic IP addresses.

Policy Opcodes

Policy FM opcodes receive a set of situational details as input and return a business policy decision as output. In general, this decision is based on the input parameters, and the policy opcode does not call any additional opcodes or access the database. However, if the business behavior of your custom policy opcode requires executing other opcodes that access data in the database, you can implement them.

Policy FM opcodes can perform any of the actions supported by FMs while deriving the data the FMs return as input. For example, in addition to the parameters passed as input, customer information can be read from the database to drive the business decision. Or an external system, such as a credit rating bureau, can be accessed to provide additional validation of a new customer during account creation.

Policy FM opcodes must conform to the standard BRM calling conventions. They receive parameters according to the input flist specification and are expected to return parameters according to the return flist specification. For detailed descriptions of the Policy FM opcodes, see the individual opcode descriptions.

Each policy opcode description explains the calling conventions and includes links to its input and output flist specifications, a detailed description of the functionality of the default implementation, and pointers to any configuration files that it uses.

Each opcode description also contains a link to the **.c** file that implements its behavior.

> **Note:**
>
> When you modify policy opcodes at the source code level, observe the transactional rules around the policy call. Because the System FMs that call the Policy FMs are responsible for the transactional integrity of BRM, they impose restrictions on certain policy opcodes. See the *Transaction Handling* section of the individual opcodes for details.

Using the Policy Opcode Source Files

You can change the default behavior of many BRM operations by using customized policy opcodes. You make your changes to the policy's **.c** file. Then, you compile and link this file with an updated Policy FM, which can be dynamically linked to any CM.

Policy opcodes are installed as executable binary files. In addition, most policy opcodes include source files that you can customize and compile. As of release 7.3, some policy opcodes do not include source files. In that case, you can create your own version of those policy opcodes.

> **Note:**
>
> In some cases, some opcodes in a single Policy FM include source code and some do not.

To create a custom version of an opcode:

1. Create a custom version of the opcode. Use the same name as the opcode supplied by BRM. For example, to create a customer version of the PCM_OP_SUBSCRIPTION_POL_SPEC_CANCEL opcode, create an opcode named PCM_OP_SUBSCRIPTION_POL_SPEC_CANCEL.

   See the opcode flist spec documentation for guidance on optional and required fields. Flist specifications are provided for all policy opcodes, including opcodes that do not have source code.

2. Edit the **fm**_*name*_**pol_config.c** file.

   For example, **fm_subscription_pol_config.c**.

3. Comment out the opcodes that you do not want to customize. If you comment out an opcode, the default functionality is used.

4. Run the **Makefile** provided with the opcode source. The **Makefile** creates a new policy library file called **fm**_*name*_**pol_custom.so,** which contains the custom policy and override policy. The entry of the custom source code must be made in the **Makefile**.

5. Open the CM **pin.conf** file and add an entry for the custom executable file. The entry must follow the entry for the default implementation. The CM reads the entries and implements the last entry it finds.

   ```
   - cm fm_module fm_subscription_pol.so fm_subscription_pol_config - pin
   - cm fm_module fm_subscription_pol_custom.so
   fm_cust_subscription_custom_config    - pin
   ```

Using the Default Implementation with Your Custom Implementation

To use the default implementation with your custom implementation:

1. Include multiple entries for the same opcode in the CM file.
   a. Open CM **pin.conf** file and add the below CM **pin.conf** entry in the last line.

      ```
      - cm fm_module ${PIN_HOME}/lib/fm_name_pol_custom${LIBRARYEXTENSION}
      fm_name_pol_custom_config - pin
      ```

      where **fm**_*name*_**pol_custom${LIBRARYEXTENSION}** denotes the custom policy library name, and **fm**_*name*_**pol_custom_config** denotes the custom policy configuration structure name.

   b. Save the CM **pin.conf** file, and restart the CM process.

   The CM loads both implementations of the policy code, the default implementation and the custom implementation. However, by default, CM runs the implementation that is configured last in the CM **pin.conf** file.
2. Use the custom opcode to call the default implementation of the policy opcode by including the CM_FM_OP_PREV_IMPL or CM_FM_OP_PREV_IMPL_BY_REF macro in your custom opcode source code.

   To add some customization to the implementations and to call the default implementation from the custom implementation, use one of the preceding functions to call the previous (default) implementation of the same policy opcode.
3. Run your custom implementation first, and then the default implementation, using the CM.

> **Note:**
>
> - In case of the CM_FM_OP_PREV_IMPL macro, a local copy of the input flist is created to call the opcode. The local copy is destroyed after the called opcode is returned.
> - In case of the CM_FM_OP_PREV_IMPL_BY_REF macro, a copy of input flist is not created. The given input flist is passed to the opcode so that the called opcode uses the input flist as a reference only and does not alter the data.

For information about the CM_FM_OP_PREV_IMPL and CM_FM_OP_PREV_IMPL_BY_REF macro, see the **cm_fm.h** file in the *BRM_home***/include** directory, where *BRM_home* is the directory in which the BRM server software is installed.

Adding a New Policy FM

A Policy FM consists of a set of opcodes and functions. The functions implement the opcodes. The functions are compiled and then linked into a shared library (the Policy FM).

For examples, see the **.c** source files for each opcode in *BRM_SDK_home***/source/sys/fm**_*category*_**pol**, where *category* is the name of the opcode category.

To create a new Policy FM:

1. Define the input and output flist specifications.

This is already done for you. Each policy opcode must conform to the input and output flist specifications referenced in the opcode reference document.

2. Write a function to implement your custom opcode.

   To do this, modify a copy of the **.c** source file provided for the corresponding default policy opcode.

   For example, to change the default behavior of the PCM_OP_CUST_POL_PREP_BILLINFO policy opcode, change the **op_cust_pol_prep_billinfo** function, which is contained in the **fm_cust_pol_prep_billinfo.c** file.

3. Create an entry in the configuration file to map the function to an opcode or if it is a new policy FM, create a configuration file to specify the opcode-to-function mapping.

   For information on creating the file, see "Creating an Opcode-to-Function Mapping File".

13

Writing a Custom Facilities Module

Learn how to add new features and functionality to Oracle Communications Billing and Revenue Management (BRM) by writing custom Facilities Modules (FMs).

Topics in this document:

- About Implementing Custom FMs
- Creating a New FM
- About Configuring a New FM into a CM
- Handling Transactions in Custom FMs
- Managing Memory in Custom FMs
- Opening a New Context in an FM
- Compiling and Linking a Custom FM
- Configuring Your New Policy FM
- Debugging FMs

For information about creating or modifying policy FMs, see "About System and Policy Opcodes".

About Implementing Custom FMs

You implement a custom FM the same way you add a custom client application in C to BRM:

- You use the same standard client libraries you use to write a custom application. These libraries are included in the BRM SDK (*BRM_SDK_home***lib**).
- You use the same coding conventions in an FM and in custom applications.
- An FM can call system opcodes and use PCM Library macros just like a client application.
- You can easily implement new features at the client application level and then migrate them into an FM after debugging.

For information on creating client applications, see "Adding New Client Applications".

FMs are configured into CMs at CM execution time as dynamically loaded libraries. When a new operation is implemented with an FM and the FM becomes part of a CM, it appears to all client applications as a fully integrated BRM operation. The custom FMs can perform checks, write log records, call functions in other FMs, call the default DMs, or call custom DMs.

Creating a New FM

Follow these procedures to create a new FM in a CM:

1. Define new opcodes. See "Defining New Opcodes".
2. Define input and output specifications for the new opcodes. See "Defining Input and Output Flist Specifications".

3. If necessary, define new storable classes and fields. See "Defining New Storable Class and Field Definitions".
4. Write a function to implement the new opcode. See "Writing a Function to Implement a New Opcode".
5. Use the **fm_post_init** function to call nonbase opcodes at CM initialization. See "Using the fm_post_init Function to Call Nonbase Opcodes at CM Initialization".
6. Write a program to map opcodes to functions. See "Creating an Opcode-to-Function Mapping File".
7. Create a shared library for the new FM. See "Creating a Shared Library for a New FM".
8. Configure the new FM as part of the CM. See "About Configuring a New FM into a CM".

Defining New Opcodes

You must define your custom opcodes, with their numbers, in a header file and run the **parse_custom_ops_fields** script on the file.

For more information on the **parse_custom_ops_fields** script, see "parse_custom_ops_fields".

BRM opcodes and their numbers are defined in the **ops/*.h** files in the *BRM_home*/**include** directory, where *BRM_home* is the directory in which the BRM server software is installed.

To pass new opcodes from a client application to a new FM, you use the PCM_OP macro.

To define a new opcode:

1. Create a header file and define your new opcodes by using this format:

   ```
   #define opcode_name_1          opcode_number_1
   ```

   For example, you might create a header file named **my_opcodes.h** with these definitions:

   ```
   #define MY_OP_SET_AGE       100001
   #define MY_OP_SET_LANGUAGE  100002
   ```

   > **Note:**
   >
   > Numbers up to 10000 are reserved for internal use.

2. Run the **parse_custom_ops_fields** Perl script by using this syntax:

   ```
   parse_custom_ops_fields -L language -I input -O output     -P java_package
   ```

   For information on the **parse_custom_ops_fields** parameters and their valid values, see "parse_custom_ops_fields".

3. For applications written with PCM C (including all MTA applications), in the **pin.conf** file for each application, create an entry using this format:

   ```
   - - ops_fields_extension_file ops_flds_ext
   ```

   Where *ops_flds_ext* is the file name and location of the memory-mapped extension file that the **parse_custom_ops_fields** script created.

4. For applications written using Java PCM, add the location of the compiled Java classes that the script generated to the CLASSPATH.

5. Make sure you include your header file both in the application that is calling the opcode and in the custom FM.

Defining Input and Output Flist Specifications

Flists are passed uninterpreted between the client application and the new FM by the PCM_OP opcode.

Follow these rules when you define flist specifications for your custom opcode:

- Your flists must conform to the BRM flist specifications.
- Both the input and output flist specifications must contain the **PIN_FLD_POID** field to identify the object being manipulated.

  For an example, see the input or output flist specifications for any of the opcode descriptions.
- The client, the custom FM, and the custom Data Manager must agree on the flist format and content, especially if you are defining new fields.

Defining New Storable Class and Field Definitions

You may have to create custom fields and storable classes for your FM. As with new opcodes and flists, the custom FM and the client application must agree on the semantics of the new fields.

If you define new storable classes, you must create a new opcode to manipulate the data stored in the new objects. Pass the new opcode to your custom FM with an flist containing the POID of the object. The FM then sends a series of basic opcodes to the DM, depending on the operation.

See "Creating, Editing, and Deleting Fields and Storable Classes" for information on defining new storable classes and fields.

You also can implement custom functionality by using **/data** objects, which contain generic fields and can be used for BLOB (Binary Large Object) processing.

Writing a Function to Implement a New Opcode

To implement a new opcode, you write a new function that calls the base system opcodes to access the DM. The new function then becomes part of a new FM shared library.

The new function you write must conform to the PCM_OP calling convention.

Use the **PCM_OP**_* reference pages as checklists and templates to determine what your custom function must implement. Pay particular attention to the input and output flists.

> **Note:**
>
> If you are adding a new function in the **fm_utils** module that can be called outside the module add a prototype of that function in the **fm_utils.h**.
>
> The following are the functions declared in the **fm_utils** module that can be modified:
>
> - **fm_utils_op_is_ancestor():** Determines if the given opcode is an ancestor of the current one. Returns **1** if true, **0** otherwise.
> - **fm_utils_op_is_parent():** Determines if the given opcode is a direct parent of the current opcode. Returns **1** if true, **0** otherwise.
> - **fm_utils_op_is_grandparent():** Determines if the given opcode is a grandparent of the current opcode. Returns **1** if true, **0** otherwise.
> - **fm_utils_op_is_entry():** Determines if the given opcode is an entry-level opcode. Returns **1** if true, **0** otherwise.
> - **fm_utils_is_subtype():** Determines if the passed type is a subtype of the passed **poid_type**. Returns **1** if true, **0** otherwise.

Using the fm_post_init Function to Call Nonbase Opcodes at CM Initialization

Using the **fm_post_init** function, custom FMs can call nonbase opcodes at CM initialization. To implement this, ensure that the **fm_post_init** function is a part of the FM shared library.

For example:

```
void
fm_post_init(int *err)
{
* errp = 0;
.
.
.
.
}
```

The CM initialization takes place in two phases:

- In the first phase, the **fm_post_init** functions are called for all the FM modules specified in the CMs **pin.conf** file. At this time, it is not possible to call nonbase opcodes from the **fm_post_init** functions.
- In the second phase, all the **fm_post_init** methods implemented by all the FM modules are called. At this time, it is possible to call nonbase opcodes from the **fm_post_init** functions.

Creating an Opcode-to-Function Mapping File

You create a configuration program, **fm_*_config.c**, to map an opcode to the function that implements it. This configuration file is read when a dynamic library is created, and the mapping information is stored in it. When a parent CM is initialized, it configures the opcode-function pairs into itself, and each child CM inherits the same configuration as its parent.

The configuration program contains an array of **struct cm_fm_config** for each opcode and its corresponding function and the **config** function at the end.

For a definition of this **struct**, see the **cm_fm.h** file in the *BRM_SDK_home*/**Include** directory.

The following example shows an opcode-to-function configuration file.

> **Note:**
>
> Include your **.h** file with your new custom opcodes.

```
 /** fm_cust_config.c 1.8 99/02/11
 *

   #ifndef lint
   static  char    Sccs_id[] = "@(#)fm_cust_config.c    1.8 11 Feb 1999 ";
   #endif

   #include                        /* for FILE * in pcm.h */
   #include "pcm.h"
   #include "ops/cust.h"
   #include "pcp.h"
   #include "cm_fm.h"

/*************************************************************
 PIN_EXPORT void * fm_cust_config_func();

struct cm_fm_config fm_cust_config[] = {
              /* opcode as a int32, function name (as a string) */
              { PCM_OP_CUST_BILLINFO,         "op_cust_billinfo" },
              { PCM_OP_CUST_CREATE_SERVICE,   "op_cust_create_service" },
              { PCM_OP_CUST_DELETE_ACCT,      "op_cust_delete_acct" },
              { PCM_OP_CUST_INIT_SERVICE,     "op_cust_init_service" },
              { PCM_OP_CUST_NAMEINFO,         "op_cust_nameinfo" },
              { PCM_OP_CUST_STATUS,           "op_cust_status" },
              { PCM_OP_CUST_VERIFY,           "op_cust_verify" },
              { PCM_OP_CUST_FINDSERV_VERIFY,  "op_cust_findserv" },
              { 0,    (char *)0 }
void *
fm_cust_config_func()
{
return ((void *) (fm_cust_config));
}
```

For example, when the CM gets a PCM_OP_CUST_BILLINFO() opcode through the PCM_OP() call from a client application, the CM looks up this table and calls the **op_cust_billinfo()** function.

> **Note:**
>
> The CM gets the name of this configuration file from its own configuration file.

Creating a Shared Library for a New FM

Each custom FM must be created as a shared library. BRM code is multi-thread (MT) safe. If you require your custom FM to be MT safe, you must make your custom FM code MT safe.

Linux: See the Linux documentation for more information on shared libraries (**LD**(1)) and making your code MT safe (**threads**(3T)).

About Configuring a New FM into a CM

New FMs are implemented as shared libraries and are dynamically linked to CMs at runtime. You add new FMs to the CM configuration file. When a new CM is started or restarted, it reads its configuration file and loads the listed FMs dynamically.

Child CM processes inherit the configuration information read by their parent CM process. Child CM processes do not read the configuration file when they are forced. You can limit the number of CMs that implement the new opcode by leaving the new opcode out of the parent CM configuration file.

You must also make sure that you use the same database number in the configuration files for your client applications, custom FM, and the DMs.

Adding a New FM Module to the CM Configuration File

The configuration file contains the names of the shared libraries that implement the base and custom opcodes. It also contains the names of the corresponding configuration files that contain the opcode-to-function mappings.

The custom shared library (**.so** on Linux) contains the functions that implement the new opcodes and the opcode-to-function mapping table struct. No opcode-to-function mapping files must be present when the CMs with the custom FMs are started because this information is already stored in the shared library. However, the name of the shared library and the mapping struct still have to be in the configuration file so that the CM can find and configure them.

For the format and description of the entries for an FM, see "Syntax for Facilities Module Entries" in *BRM System Administrator's Guide*.

Use the entries for the system FMs in the default CM configuration file in *BRM_SDK_home***/sys/cm** as an example to add your custom FM entries.

Initializing Objects for Multiple Processes

In the CM configuration file, you can specify an initialization function that is called when the CM loads an FM or Policy FM. This function initializes objects that are called in multiple processes or threads.

1. Create a new file named ***_init.c** (for example, **fm_term_pol_init.c**) in the appropriate FM or Policy FM directory (for example, **fm_term_pol** directory).
2. Add the new file to the **Makefile.user** file.
3. Implement an initialization function in the ***_init.c** file.

   Example pseudo code:

```
pin_flist_t *global_flistp = NULL;
extern void
fm_term_pol_init(int32 *errp)
```

```
{
      global_flistp = read from custom objects;
       *errp = PIN_ERR_NONE;
}
```

> **Note:**
>
> In the example above, **global_flistp** is allocated in the CM master process or thread. When a child process is created, **global_flistp** is duplicated and still available on the child process.

4. Add the initialization function to the CM configuration file, using the following example:

```
- cm fm_term_pol ../../lib/fm_term_pol.lib  fm_term_pol_config_func
fm_term_pol_init pin
```

Handling Transactions in Custom FMs

All policy operations conform to the rules for application-level transactions.

See the *Transaction Handling* section of the individual opcodes for details.

If a read-write or read-only transaction is open when a policy operation is performed, all data read as part of the operation will be consistent with the state of the database when the transaction is opened. This guarantees that the data used by this operation is consistent with related data used by other operations in the transaction.

If this operation is called when a transaction is not already open, the operation is performed without transactional control.

> **Caution:**
>
> Policy operations must not modify object data.

Custom FM code is responsible for starting, stopping, and committing transactions as required depending on the semantics of the opcode. Each base opcode must be surrounded by transactions, unless a transaction is already open when the opcode is called.

All other system opcodes, except for PCM_OP_PYMT_CHARGE and the password opcodes, start transactions if none are open when they are called.

Custom FM transactions must conform to the transaction specifications of the PCM context management functions. See "Context Management Opcodes" for the rules to follow.

For information on system transaction handling, see the Transaction Handling sections of each opcode description.

In your custom FM, you can check for open transactions. See the **fm_generic_opcode.c** file in *BRM_SDK_home***/templates/fm_template** for sample code to use.

A custom FM can call other FMs by using the Portal Communications Module (PCM) API. Also, custom FMs use the same PCM API used to call other FMs and Data Managers.

> **Note:**
>
> Calls to the Data Manager are made with base opcodes.

Managing Memory in Custom FMs

To manage memory in your custom FMs, follow these rules:

- Always use **pin_malloc()**, **pin_free()**, **pin_realloc()**, and **pin_strdup()** for the **get(), set(), take(),** and **put()** operations on flists.
- Use the standard routines-**malloc(3C)**, **free(3C)**, **realloc(3C)**, and **strdup(3C)**-for other memory operations.

Opening a New Context in an FM

To open a new context in an FM, use (**pin_flist_t**)**NULL** for **in_flistp** in the PCM_CONTEXT_OPEN call.

See the **sample_app.c** and **sample_search.c** code examples for more information.

Compiling and Linking a Custom FM

After you define or modify the policy opcodes, create a shared library for the new Policy FM by using the **Makefile** included in each Policy FM source directory. Also, copy the contents from the *BRM_home***/PortalDevKit/include** directory to the *BRM_home***/include** directory.

Enter **make** to create the **.so** or **.a** file.

Use the libraries in *BRM_SDK_home***/lib** for linking.

See the sample **Makefile** for *BRM_SDK_home***/source/templates/fm_template/fm_generic_opcode.c**. The main routine for a custom opcode should look similar to the **op_generic** function in the **fm_generic_opcode.c** file. The calling parameters and their types are required.

The following example shows the list of **include** files required. Make sure you include your own header file containing your new opcodes (**custom_opcodes.h** in the following example). This example also shows skeleton code for the new opcode, the **cm_fm_config** struct, and the new function:

```
#define PCM_OP_FM_SAMPLE_LOOPBACK               999999

struct cm_fm_config fm_bill_config[] = {
         { PCM_OP_FM_SAMPLE_LOOPBACK,           "op_fm_sample_loopback" },
         { 0,    (char *)0 }
};

#include "pcm.h"
#include "cm_fm.h"
#include "pin_errs.h"
#include "pinlog.h"
#include "custom_opcodes.h"

op_fm_sample_loopback(connp, opcode, flags, i_flistp, r_flistpp, ebufp)
         cm_nap_connection_t     *connp;
```

```
        int32                   opcode;
        int32                   flags;
        pin_flist_t             *i_flistp;
        pin_flist_t             **r_flistpp;
        pin_errbuf_t            *ebufp;
{
        PIN_ERR_CLEAR_ERR(ebufp);

        /*************************************************************
         * Check for errors.
         *************************************************************/
        if (opcode != PCM_OP_FM_SAMPLE_LOOPBACK) {
                pin_set_err(ebufp, PIN_ERRLOC_FM,
                        PIN_ERRCLASS_SYSTEM_DETERMINATE,
                        PIN_ERR_BAD_OPCODE, 0, 0, opcode);
                PIN_ERR_LOG_EBUF (PIN_ERR_LEVEL_ERROR,
                        "op_fm_sample_loopback", ebufp);
                return;
         }

         /*************************************************************
          * Return a copy of our input flist.
          *************************************************************/
         *out_flistpp = PIN_FLIST_COPY(in_flistp, ebufp);

         return;
}
```

Configuring Your New Policy FM

The shared library (**.so** on Linux) for the Policy FM must be included in each CM where its functionality is needed. Whenever a new CM is started or restarted, the CM reads its configuration file and loads the listed System FMs and Policy FMs dynamically. For more information, see "Writing a Custom Facilities Module".

Configure the new FM as part of the applicable CMs by adding the new Policy FMs to the CM configuration file.

For information on the format of the configuration file entries, see the CM configuration file.

Debugging FMs

You can debug custom and policy FMs using BRM SDK and standard programming tools.

To debug FMs and policy opcodes, you need BRM SDK, access to a functioning BRM server, and the programming tools supported by BRM SDK on your platform. See "About BRM SDK" for installation instructions and other information about BRM SDK.

For an overview of the connections required to test an FM, see "Testing New or Customized Policy FMs".

The primary way of debugging an FM is attaching to a running CM.

Various debugging tools are available on the supported operating systems:

- On Linux, you can use the **gdb** debugger.

14

Writing a Custom Data Manager

Learn how to access data in a custom data storage or legacy storage system by creating a new Oracle Communications Billing and Revenue Management (BRM) Data Manager (DM).

You must create your custom objects before creating a new Data Manager.

Topics in this document:

- About Adding a Custom Data Manager
- Understanding the Data Manager Interface
- Creating a Custom Data Manager

For more information, see "Creating Custom Fields and Storable Classes".

About Adding a Custom Data Manager

DMs provide an object model on top of different underlying storage models. There is a standard interface, Storage Manager (SM), between the generic DM code and the various underlying storage access codes. This section describes the SM interface, which you customize to create a custom Data Manager (DM).

You can add a new DM to the BRM system for the following reasons:

- To map object operations to a different storage paradigm. See "About Mapping Objects to Alternate Storage Mechanisms".
- To interface to legacy systems. Object operations can be mapped to online protocols or other methods of interfacing to the legacy systems. See "About Adding Interfaces to Legacy Systems".
- To automatically manage queues and avoid starvation for operations. This works well with legacy systems where high latency may occur.

About Mapping Objects to Alternate Storage Mechanisms

BRM is shipped with a standard SM, which provides an interface for mapping object operations to any storage paradigm that you require. BRM views the data at the object operation level, so there is no effect on the rest of the system when a data set is managed by a custom SM. This flexibility lets you use highly specialized storage paradigms, such as an indexed file system.

About Adding Interfaces to Legacy Systems

BRM allows transparent integration with legacy systems. Operations that are routed to a legacy system for execution appear as object manipulations within the BRM system. The interface to the legacy system is written with the same client APIs you use to create custom applications. Only the custom SM, which is the legacy translation module, is aware that the object operations are being translated. The fact that the operations are not performed within the BRM system is transparent to all other modules in the system.

You can integrate any type of legacy storage system with BRM. Any type of object operation can be defined and sent to the custom SM for translation to the legacy system.

Understanding the Data Manager Interface

To build a new DM, you must understand the following:

- Calling Conventions
- Data Manager Memory Model
- Function Entry Points

Calling Conventions

Your custom DMs can be called only from the base opcodes. The FM opcodes call other underlying opcodes, which in turn call base opcodes which are run by the DMs. The **ops/base.h** header file must be included in your application unless the application uses an FM opcode. FM opcodes already include the base opcode header file.

When you create a custom DM, you must implement the base opcodes or a subset of the base opcodes that your DM requires to provide the functionality you want. Each of the DMs included with BRM uses a different Implementation of a base opcode depending on the DM and the storage system it interacts with. For example, the base opcode PCM_OP_SEARCH is implemented differently for **dm_oracle** and **dm_ldap**.

For details, see the descriptions of the PCM_OP_SEARCH opcodes in "Base Opcodes" in *BRM Opcode Guide*.

Data Manager Memory Model

The DM uses shared memory to pass data back and forth between the BRM front-end and back-end (Storage Manager) processes. The DM uses a queuing-based memory management model. For more information about queuing based memory management, see "About Queuing-Based Processes" in *BRM System Administrator's Guide*.

The DM and QM are separate processes, so external libraries do not have to be multi-thread safe.

For an example of queuing-based processes, see "Example of Queuing in a Client-to-CM Connection" in *BRM System Administrator's Guide*.

Function Entry Points

The routines described in this section provide the entry points for your custom DM. You must name and define the entry points exactly as shown in the following list. Different underlying storage modules are dynamically linked when you use the **dm_sm_obj** keyword in the DM configuration file. If this dynamic linking does not work, link the DMs directly using a DM-specific **makefile**.

dm_if_init_process()

This routine sets up the initial process. It is called when a child DM is started, for example, to connect to a database. It reads in the configuration information from the DM configuration file. This routine uses the following syntax:

```
void
dm_if_init_process(struct dm_sm_config *confp, int32 *errp)
```

dm_if_process_op()

This routine processes an operation that comes from the CM. This is implemented in the backend. This routine uses the following syntax:

```
void
dm_if_process_op(
      struct dm_sm_info *dsip,
       int32pcm_op,
       int32pcm_flags,
       pin_flist_t*in_flistp,
       pin_flist_t**out_flistpp,
       pin_errbuf_t*ebufp)
```

dm_if _terminate_connect()

When the CM or the DM is disconnected, this routine is called to clean up the CM connection, for example to rollback a transaction in progress. It is called only when a SIGQUIT signal is sent to the DM main process.

This routine uses the following syntax:

```
void
dm_if_terminate_connect(
       struct dm_sm_info *dsip,
    int32 *errp)
```

dm_if_terminate_process()

If a custom DM is stopped with a SIGQUIT signal resulting from a **kill -QUIT** *pid*' command in the stop script, this routine is called when the DM process is terminated. This routine uses the following syntax:

```
void
dm_if_terminate_process(int32 *errp)
```

Argument Descriptions

Table 14-1 describes the arguments used in the DM entry-point routines:

Table 14-1 Arguments Used in DM Entry-Point Routines

Argument	Description
confp	Pointer to the structure that contains the information about the DM to SM configuration. It is passed in to **dm_if_init_process()**. The structure contains the following elements: • **be_id**: ID number of the SM starting from 0. • **sm_shm_size**: Size of the shared memory allocated to this SM. A value of 0 means there is no shared memory allocated to the SM. • **sm_shm_base**: Base shared memory allocated if the **sm_shm_size** is not 0. See **dm_sm.h** for more information.

Table 14-1 (Cont.) Arguments Used in DM Entry-Point Routines

Argument	Description
dsip	Pointer to the structure that contains the information about the DM to SM connection such as the connection state. The structure contains two sets of information, public and private, relevant to the underlying code. Public information: • **poidp**: Pointer to the POID of the input flist. • **who**: Pointer to the ID of the sender. • **trans_flag**: contains the transaction flags Private information: • **pvti**: An **int** for private use. • **pvtp**: Pointer to the SM private area. See **dm_sm.h** for more information.
pcm_op	A base PCM opcode. Only the opcodes **PCM_OP_CREATE_OBJ** through **PCM_OP_TRANS_COMMIT** defined in the *BRM_home***include\ops\base.h** are supported in the DM. *BRM_home* is the directory in which the BRM server software is installed.
pcm_flags	Bit-mask flags that you can set for operations. For information on different flags, see *BRM_home***include\pcm.h.**
in_flistp	Input flist pointer. It can be used as it is in **pin_flist_xxx** functions.
out_flistpp	Output flist pointer. It can be used as it is in **pin_flist_xxx** functions.
ebufp	Pointer to an error buffer structure. For a definition of the structure, see **pcm.h**. **ebufp** pointing to a **pin_err** will be **PIN_ERR_NONE** on entry. Use a different value to indicate an error. Also, if there is an error, specify the rest of the values or clear them.

Creating a Custom Data Manager

Use the *BRM_SDK_home*/**source/templates/dm_temp/dm_generic.c** program as a template for your new DM. The **dm_generic.c** program just echoes any flist sent into it, but it contains all the elements needed for a new DM.

Follow these programming guidelines when writing a new DM.

- Creating a New Data Manager
- Managing Memory
- Handling Errors
- Configuring Your CM to Use the Custom DM
- Editing Your Custom Opcodes to Access the Custom DM

Creating a New Data Manager

To create a new DM, perform the following tasks:

1. Writing, Compiling, and Linking a Custom DM
2. Configuring Your Custom DM
3. Starting and Stopping Your Custom DM

Writing, Compiling, and Linking a Custom DM

1. Write the new DM code, using *BRM_SDK_home***/source/templates/dm_template/dm_generic.c** as a template.
2. Run the make file that accompanies **dm_generic.c** to compile the custom DM. Edit the make file to refer to the new source file.

Configuring Your Custom DM

1. Create a directory for your custom DM and copy the compiled **.so** file to that directory.
2. Copy the configuration file (**pin.conf**) from *BRM_SDK_home***/sys/dm** to the new directory.
3. Edit the **pin.conf** file to refer to the new custom DM.

   The file includes information about changing its entries.

Starting and Stopping Your Custom DM

This section provides the steps for creating Start and Stop scripts. In this example, the custom DM is called **dm_new**.

1. Go to the *BRM_home***/bin** directory:

   ```
   cd BRM_home/bin
   ```

2. Copy the DM start and stop scripts to the *BRM_home***/bin/** directory. For example, if BRM is using the Oracle DM:

   ```
   cp start_dm_oracle start_dm_new
   cp stop_dm_oracle stop_dm_new
   ```

3. Create a symbolic link from **dm** to the **dm_new** to distinguish custom DM processes from other DM processes:

   ```
   ln -s dm dm_new
   ```

4. Edit the following entries in the start script to reference your custom DM and save it:

   ```
   start dm_new

   DM=BRM_home/bin/dm_new

   DMDIR=BRM_home/sys/dm_new

   LOGDIR=BRM_home/var/dm_new

   DMLOG=${LOGDIR}/dm_new.log

   DMPID=${LOGDIR}/dm_new.pid
   ```

5. Edit the stop script.

   a. Change the following entries to reference your custom DM:

   ```
   stop dm_new

   DM=dm_new

   LOGDIR=BRM_home/var/dm_new

   DMPID=${LOGDIR}/dm_new.pid
   ```

b. Change the **kill** entry to include the QUIT signal:

```
kill -QUIT `cat ${DMPID}`
```

c. Save the stop script.

6. Start your custom DM and verify that the scripts are working:

```
start_dm_new

ps -ef | grep new

...
cd BRM_home/var/dm_new

more dm_new.pinlog
D Thu Mar 12 15:43:51 1999  trainsun10  dm:8241  dm_main.c(1.82):1508

DM dm_name set to "-"
```

Managing Memory

To allocate storage from the shared memory segment instead of the stack, use **pin_malloc**(), **pin_strdup**(), **pin_free**(), and **pin_realloc**() routines in each custom DM. This allows the front-end and back-end processes to pass data between them, since the only common portion of their address spaces is the shared memory segment.

Use these routines to manage flists and to do **GET**/**SET** or **PUT**/**TAKE** flist operations.

For managing memory that is not related to flists, use the standard Linux versions of the memory-management routines: **malloc**(3C), **strdup**(3C), **free**(3C), and **realloc**(3C).

To allocate memory in the shared memory area:

1. Use the Linux **pin_malloc**(), **pin_strdup**(), or **pin_realloc**() routine.
2. Use the **pin_free**() routine.

Handling Errors

You can use one or a combination of the following ways to return failure status to the application.

- You can set an error condition in the **ebuf**, which is passed back as the **ebuf** to the application that calls **pcm_op**().
- You can use **PIN_ERR_NONE** in the **ebuf** and use a field on the return flist to identify the reason for failure.

When writing a custom DM, follow the BRM conventions for error handling:

- Set errors in **ebuf**.
- Set errors in fields on the return flist so that applications or code that calls it can read the error.

Configuring Your CM to Use the Custom DM

In your CM configuration file, add the following entries:

- **dm_pointer**, which specifies where to find your DM. Each pointer has three values:
  - Database number, such as 0.0.0.4

  - IP address or host name of the computer running your custom DM
  - Port number of the DM service

```
- cm  dm_pointer  0.0.0.1  test_machine  11950
```

- The database number for the custom FM to access. The entry has the name of the FM accessing the custom DM, the name of the configuration entry, and the following values for the database number, which must have the same format as the POID:
  - The database number, such as 0.0.0.2 in the example below. This value is required and must be the same as that of the **dm_db_no** in the configuration file of the custom DM.
  - The service type such as **/cc_db** for the credit card processing service. This can be any meaningful text string to identify the custom database and is a placeholder for the POID format.
  - The ID of 0, which is an arbitrary value needed as a placeholder for the POID format.

An example of these entries in the configuration file is as follows:

```
- fm_bill  cc_db  0.0.0.2  /_cc_db  0
```

Editing Your Custom Opcodes to Access the Custom DM

If you have written custom opcodes, you must edit them to access your custom DMs.

For information on editing your custom opcodes, see "About System and Policy Opcodes".

15

Creating Custom Fields and Storable Classes

Learn how to create custom fields and storable classes using the Oracle Communications Billing and Revenue Management (BRM) Software Development Kit (SDK) opcodes, Storable Class Editor (part of Developer Center), and utilities.

Topics in this document:

- Creating, Editing, and Deleting Fields and Storable Classes
- About BRM SDK Opcodes
- Converting Storable Class Files from Previous Versions
- Deploying Custom Fields and Storable Class Definitions
- Adding Fields to /config Objects

For a description of the storable class structure and a list of predefined storable classes, see "Understanding Flists".

Creating, Editing, and Deleting Fields and Storable Classes

To create, edit, and delete fields and custom storable classes, first determine the data that you want them to contain. You then enter the information into Storable Class Editor, using BRM conventions for naming and formatting.

To manage field and storable class specifications in Storable Class Editor:

1. Enable changes to the data dictionary. See "Modifying the pin.conf File to Enable Changes".
2. Create your custom fields. See "Creating Custom Fields".
3. Create your custom storable classes. See "Creating Custom Storable Classes".
4. Make your custom fields and storable classes available to BRM by generating source and header files. See "Making Custom Fields Available to Your Applications".

For more information, see "About Defining Storable Classes" and "Storable Class Naming and Formatting Conventions".

Modifying the pin.conf File to Enable Changes

Before you can add or change fields and storable classes, you must make the data dictionary writable by editing the Data Manager (DM) configuration (**pin.conf**) file.

To make the data dictionary writable, perform the following for each database in your system:

1. Open the Oracle DM configuration file (*BRM_home*/**sys/dm_oracle/pin.conf**) in a text editor. (*BRM_home* is the directory in which the BRM server software is installed.)
2. To enable field creation in the data dictionary, set the following entry to **1**:

```
- dm dd_write_enable_fields 1
```

3. To enable the creation, editing, or deletion of custom storable classes in the data dictionary, set the following entry to **1**:

```
- dm dd_write_enable_objects 1
```

Increasing the Size of the CM Cache for the Data Dictionary

If your data dictionary contains a lot of data, you might need to increase the space allocated to it in the CM cache.

To increase the size of the CM cache for the data dictionary:

1. Open the Connection Manager (CM) configuration file (*BRM_home***/sys/cm/pin.conf**).
2. Increase the *cache_size* in the following entries:

```
- cm_cache cm_data_dictionary_cache number_of_entries, cache_size, hash_size

- cm_cache fm_utils_data_dictionary_cache number_of_entries, cache_size, hash_size
```

3. Save the file.
4. Stop and restart the CM.

Using DDL when Updating the Data Dictionary Tables

You can configure the DM to run Data Definition Language (DDL) when updating object types in the data dictionary tables. This ensures that database objects are mapped to the correct tables.

To specify whether DDL is used when updating the data dictionary tables:

1. Open the Oracle DM configuration file (*BRM_home***/sys/dm_oracle/pin.conf**) in a text editor.
2. Set the **sm_oracle_ddl** entry to one of the following:
   - **0** to not run DDLs when updating object types in the data dictionary.
   - **1** to run DDLs when updating object types in the data dictionary.

```
- dm sm_oracle_ddl 1
```

3. Save and close the file.

Creating Custom Fields

When you create a new field in Storable Class Editor, it is committed to the data dictionary when you click **OK** in the New Field dialog box. You can't delete a field after it has been committed.

1. In Storable Class Editor, choose **File - New - Field**.
2. In the **Field Name** box, enter a unique name for the new field.
3. In the **Type** list, choose a field type in the list.
4. (Optional) In the **Description** box, enter text to define the purpose of the field.
5. (Optional) In the **Field ID** field, change the automatically assigned ID number.

   Table 15-1 lists the field ID ranges for Oracle-only use and customer use.

Table 15-1 BRM Field ID Restrictions

Field ID Range	Reserved For
0 through 9999	Oracle use only
10,000 through 999,999	Customer use
1,000,000 through 9,999,999	Oracle use only
Over 10,000,000	Customer use

6. Click **OK** to add the field to the data dictionary.

Creating Custom Storable Classes

You can create custom base classes and subclasses. Classes are not saved to the data dictionary until you commit them.

> **Caution:**
>
> When you create a subclass, the total number of tables for the parent class and its subclasses cannot be greater than 64. For example, if you create tables for storable class **/a** and its subclasses **/a/b** and **/a/b/c**, the total number of tables for all three storable classes must be less than or equal to 64:
>
> **/a** tables + **/a/b** tables + **/a/b/c** tables <= 64 tables

1. In Storable Class Editor, choose **File - New - Class**.
2. In the **Class Name** field, enter the class name in the format **/classname** for base classes and **/classname/subclass** for subclasses.

> **Note:**
>
> You cannot use /publish as a custom storable class name as Oracle DM uses it to differentiate publishing operations from normal DML operations. /publish is a reserved keyword.

3. (Optional) In the **Label** field, enter a name for the storable class.
4. (Optional) In the **Description** field, enter text to define the purpose of the storable class.
5. (Optional; base classes only) In the **Sequence Start** field, enter a number to designate how objects in this class should be numbered.
6. (Optional; base classes only) In the **Table Name** field, optionally change the default SQL table name suggested by Storable Class Editor. You should accept the default table name unless your business logic requires you to change it.
7. (Optional; base classes only) In the **Storage Specifications** field, enter storage specifications for the database you are using.

   For Oracle databases, parameters include tablespace, and initial extent size. See "Database Configuration and Tuning" in *BRM Installation Guide*.
8. Click **OK**.

The storable class opens in a Class Definition window. The class name appears in the title bar.

9. Select the root icon of the new class.
10. In the Properties window, choose values for the **Read Access** and **Write Access** properties.
11. Add fields to the class by dragging field icons from the Class Browser or Field Browser.

> **Note:**
>
> - Each class must have the PIN_FLD_ACCOUNT_OBJ field. If you do not include this field, it is automatically inserted when the class is created.
> - The maximum allowed number of fields/columns for a BRM class is 128. If a class exceeds this limit, PCM_OP_CREATE_OBJ will fail and throw an error.

12. Choose **File - Commit New Class** to commit the class to the data dictionary.
13. If you created a new subclass *and* if the base class is partitioned, you must run the **partition_utils** script with the **-n** parameter to ensure that the new subclass uses the same partitioning layout as its base class.

    See "Partitioning Database Tables" in *BRM System Administrator's Guide*.

Making Custom Fields Available to Your Applications

After you create custom fields and classes, you must make them available to applications. The first step is to use Storable Class Editor to create Java source files and a C header file. The steps that follow depend on whether your applications are written in C or Java. For more information about using custom fields in Java applications, see "Using Custom Fields in Java Applications".

> **Note:**
>
> Developer Center is a Java application. To ensure that custom fields are displayed properly in flists in Object Browser and Opcode Workbench, you must follow the procedures for making fields available to Java applications.

1. In Storable Class Editor, choose **File - Generate Custom Fields Source** to create source files for your custom fields.

   Storable Class Editor creates a C header file called **cust_flds.h**, a Java properties file called **InfranetPropertiesAdditions.properties,** and a Java source file for each custom field.
2. For applications written in PCM C or PCM C++, perform these steps:
   a. Run the **parse_custom_ops_fields.pl** Perl script on the **cust_flds.h** file created by Storable Class Editor. Use this syntax:

      ```
      parse_custom_ops_fields -L language -I input -O output
      ```

      For information on the parameters of **parse_custom_ops_fields** and their valid values, see "parse_custom_ops_fields".

b. In the **pin.conf** file for applications that must access these fields, including **testnap** and other utilities, create an entry using the format shown below. Replace *cust_flds* with the file name and location of the memory-mapped extension file that the **parse_custom_ops_fields** script created.

```
- - ops_fields_extension_file cust_flds
```

> **Note:**
>
> Do not add more than one **ops_fields_extension_file** entry. The custom fields source file and the extension file that results from it contain information about all the custom fields in the data dictionary, so a single reference to that file is sufficient.

c. Include the **cust_flds.h** header file in the applications and in the FMs that use the fields.

> **Note:**
>
> Default BRM fields are defined with their numbers in the **pin_flds.h** file in the *BRM_home*/**include** directory. While it is possible to add custom fields directly to **pin_flds.h**, you should not do so. Placing custom field definitions in the separate **cust_flds.h** file allows you to upgrade to new releases without having to edit **pin_flds.h**.

3. For applications written using Java PCM, including Developer Center, perform these steps:
   a. Copy the contents of the **InfranetPropertiesAdditions.properties** file and paste it into the **Infranet.properties** file for your application.
   b. Compile the source files you created in step 1.
   c. (Optional) Jar the compiled classes.
   d. In the CLASSPATH, add the location of the **JAR** files or compiled Java classes.

About BRM SDK Opcodes

The BRM SDK opcodes allows you to create, modify, delete, or retrieve storable class and field specifications without the use of BRM Storable Class Editor.

For information about BRM SDK, see "About BRM SDK".

To manage field and storable class specifications with BRM SDK opcodes:

1. Enable changes to the data dictionary. See "Modifying the pin.conf File to Enable Changes".
2. Create, edit, or delete your custom fields. See "Using BRM SDK Opcodes to Manage Storable Classes".
3. Create, edit, or delete your custom storable classes. See "Using BRM SDK Opcodes to Manage Field Specifications".
4. Make your custom fields and storable classes available to BRM by generating source and header files. See "Making Custom Fields Available to Your Applications".

Using BRM SDK Opcodes to Manage Storable Classes

Use the following BRM SDK opcodes to manage storable class specifications:

- To create or modify a storable class specification, use PCM_OP_SDK_SET_OBJ_SPECS. See "Creating and Modifying Storable Classes".
- To retrieve a storable class specification, use PCM_OP_SDK_GET_OBJ_SPECS. See "Retrieving Storable Class Specifications".
- To delete a storable class specification, use PCM_OP_SDK_DEL_OBJ_SPECS. See "Deleting Storable Class Specifications".

Creating and Modifying Storable Classes

Use the PCM_OP_SDK_SET_OBJ_SPECS opcode to create or modify a storable class. This opcode creates or modifies storable classes in the data dictionary of all databases in your BRM system.

> **Caution:**
>
> If you change a storable class after it has been instanced and populated with data, you will corrupt your database.

> **Note:**
>
> Instead of using this opcode, it's safer and more reliable to create or modify storable class specifications by using the Storable Class Editor in Developer Center.

PCM_OP_SDK_SET_OBJ_SPECS takes the following as input:

- POID
- Storable class name
- Storable class type

> **Note:**
>
> The Portal object ID (POID) is the only mandatory field on the input flist. However, to create a storable class specification, you must at least specify the storable class name and type. To specify fields for the storable class, add a PIN_FLD_OBJ_ELEM array for each field.

If the transaction is successful, PCM_OP_SDK_SET_OBJ_SPECS returns these values:

- The POID of the newly created or modified storable class.
- A results array containing an SQL description of any table changes, with one array for each change.

Retrieving Storable Class Specifications

Use the PCM_OP_SDK_GET_OBJ_SPECS opcode to retrieve one or more storable class specifications. This opcode retrieves all storable class specifications specified on the input flist. When no storable classes are specified, this opcode returns all storable class specifications in the BRM database.

> **Note:**
>
> You can retrieve specific levels or types of objects by using the ***** wildcard character.

PCM_OP_SDK_GET_OBJ_SPECS returns the following, depending on the success of the transaction:

- When successful, this opcode returns specifications for the specified storable classes, or all storable class specifications if the input flist does not specify a storable class.
- When the opcode doesn't exist in the database, the opcode returns PIN_ERR_BAD_ARG.

Deleting Storable Class Specifications

Use the PCM_OP_SDK_DEL_OBJ_SPECS opcode to delete storable class specifications from the data dictionary of all databases in your BRM system.

> **Note:**
>
> The opcode deletes data from the data dictionary only. To drop the actual table that was created by PCM_OP_SDK_SET_OBJ_SPECS, you must drop it manually.

> **Caution:**
>
> If you delete a storable class that has already been instantiated, you will corrupt your database. For example, never delete the **/account** object. Because of this danger, we recommend that you *do not* use this opcode on a production system.

PCM_OP_SDK_DEL_OBJ_SPECS returns the following, depending on the success of the transaction:

- When the storable class does not exist, the opcode returns PIN_ERR_BAD_ARG_EBUF.
- When successful, the opcode returns these values:
  - The POID of the deleted object.
  - A results array containing an SQL description of any table changes; one array for each change.

Using BRM SDK Opcodes to Manage Field Specifications

Use the following BRM SDK opcodes to manage field specifications:

- To create or modify a field specification, use PCM_OP_SDK_SET_FLD_SPECS. See "Creating and Modifying Field Specifications".
- To retrieve a field specification, use PCM_OP_SDK_GET_FLD_SPECS. See "Retrieving Field Specifications".
- To delete a field specification, use PCM_OP_SDK_DEL_FLD_SPECS. See "Deleting Field Specifications".

Creating and Modifying Field Specifications

Use the PCM_OP_SDK_SET_FLD_SPECS opcode to create or modify field specifications. This opcode creates or modifies the specified field specifications in the data dictionary of all databases in your BRM system.

> **Caution:**
>
> If you change specifications for fields that have already been instantiated, you will corrupt your database.

PCM_OP_SDK_SET_FLD_SPECS takes the following as input:

- Partial POID (database number plus **/dd/fields**)
- Field name
- Field type

> **Note:**
>
> The POID is the only mandatory field on the input flist. However, to implement the field, you must at least specify the field name and type.

PCM_OP_SDK_SET_FLD_SPECS returns the following, depending on the success of the transaction:

- If successful, the opcode returns the POID of the created or modified data dictionary field.
- If the opcode cannot create or modify the field, the opcode returns the field's POID, along with the PIN_FLD_ACTION field set to **NOOP**.

Retrieving Field Specifications

Use the PCM_OP_SDK_GET_FLD_SPECS opcode to retrieve one or more field specifications. This opcode retrieves all field specifications specified on the input flist. When no fields are specified, this opcode returns all field specifications in the BRM database.

> **Note:**
>
> Returning all field specifications can take a long time.

Deleting Field Specifications

Use the PCM_OP_SDK_DEL_FLD_SPECS opcode to delete field specifications. This opcode deletes the specified field specification from the data dictionary of all databases in your BRM system.

> **Caution:**
>
> If you delete specifications for fields that have already been instantiated, you will corrupt your database. For example, never delete PIN_FLD_POID from a base BRM system. Because of this danger, we recommend that you *do not* use this opcode on a production system.

PCM_OP_SDK_DEL_FLD_SPECS takes the following as input:

- Partial POID (database number plus **/dd/fields**)
- Name of the field to delete

If successful, this opcode returns the POID of the deleted field specification.

Converting Storable Class Files from Previous Versions

If you used Developer Workshop in the past to create storable classes, you may have saved storable class definitions as files in **.PSC** format. To use these files in Storable Class Editor, you must convert them to **.SCE** format before you can open them.

You use the PSC Converter application to convert files. This application is installed along with the rest of Developer Center.

1. From the start options, choose **PSC to SCE Converter**, which is under **Portal**.
2. Click the **Select File** button and navigate to the **.PSC** file you want to convert.

   The Save As field automatically fills in with the same path. The filename is changed to include the **.SCE** extension. (If a **.SCE** file of that name already exists, the Save As field is left blank. Enter a different path and filename or click New File to choose an existing file to overwrite.)
3. Click **Convert**.

   A dialog box confirms the creation of the new **.SCE** file.

Deploying Custom Fields and Storable Class Definitions

You deploy your custom fields and storable classes by using the **pin_deploy** command-line utility. The **pin_deploy** utility exports and imports field and storable class definitions from one BRM database to another, such as from your development environment to your production environment.

The **pin_deploy** utility is available on all BRM platforms, can be scripted, and can use **stdin** and **stdout**. It has several modes of operation to ensure atomic operations and consistency. **pin_deploy** provides the following advantages:

- Streamlines the process of putting all storable class and field definitions into source code management
- Enables you to print out a storable class or field definition for review
- Reduces the possibility of damaging the BRM production database data dictionary

The **pin_deploy** utility uses PODL (Portal Object Definition Language) to export and import field and storable class definitions. PODL is a text-based definition language that represents fields and storable classes. Using this language, **pin_deploy** can:

- Extract storable class and field definitions from any BRM database on any platform and produce a human-readable PODL file.
- Read PODL files and use the files to load storable class and field definitions into any BRM server on any platform.

You can use the following command-line options in **pin_deploy**:

- **field** extracts field definitions.
- **class** extracts storable class definitions.
- **verify** connects to BRM database, accepts PODL commands, determines what changes would be made, and reports back any conflicts.
- **create** connects to a BRM database, attempts to create storable classes and fields according to PODL, succeeds if there are no conflicts, or reports conflicts.
- **replace** connects to a BRM server; attempts to create storable classes and fields according to PODL, and succeeds even if there are conflicts. It overwrites and replaces any storable classes that are already present.

In all cases, the entire PODL file is imported. If the entire file cannot be loaded correctly, nothing from the file is loaded. For example, if a storable class is loaded that includes custom fields, those custom fields must exist in the data dictionary or in the PODL file for the storable class to load.

> **Note:**
>
> Before you deploy your custom storable classes and fields, verify that you have enough space in the BRM database for the new storable classes. If you run out of space during deployment, the new storable classes might be in an inconsistent state.

See "pin_deploy" for more information and a complete list of options.

Extracting Field and Storable Class Definitions with pin_deploy

> **Note:**
>
> You must run **pin_deploy** from the *BRM_home***/sys/test** directory because it must be run with the **root.0.0.0.1** role.

To extract field definitions, enter a command with the following syntax:

```
pin_deploy field [-cp] [field_name1] [field_name2]
```

To extract storable class definitions, use the following syntax:

```
pin_deploy class [-smncp] [class_name1] [class_name2]
```

Importing Storable Class Definitions with pin_deploy

To import storable class definitions into a BRM database, you must use PODL files that include the interface and corresponding implementation definitions.

> **Note:**
>
> - The **pin_deploy** utility cannot determine the space requirement in the BRM database. If you run out of disk space before the deployment is complete, you must manually drop the tables that were created, make more space, and try again.
> - You must run **pin_deploy** from the *BRM_home***/sys/test** directory because it must be run with the **root.0.0.0.1** role.

1. Add up the implementation definitions (for example, **initial clause**) of the PODL files you want to import to verify that you have enough disk space.

   These lines start with this text:

   ```
   SQL_STORAGE =
   ```

2. Run **pin_deploy** in the **verify** mode to determine the changes that will be caused by importing new field and storable class definitions and to verify that there are no conflicts.

   ```
   pin_deploy verify [file_name1] [file_name2]
   ```

3. To commit new definitions to BRM, run **pin_deploy** in one of the following modes.
   - **create**: Preserves old storable class and field definitions that conflict with new ones
   - **replace**: Copies all storable class and field definitions, including those that conflict with old definitions

     ```
     pin_deploy replace [file_name1][file_name2]
     ```

Adding Fields to /config Objects

Many BRM features use **/config** objects to store business configuration information in the database.

In some cases, you may need to add or replace fields in **/config** objects.

You can use either Developer Center or **testnap** to make changes to **/config** objects. The general procedure is to display the current contents of the object, write an flist that contains the fields to add, and then write the flist to the object.

The following two sections provide examples for both Developer Center and **testnap**. Depending on the nature of the **/config** object, the exact procedure you use may be different from the examples.

Using Developer Center to Modify /config Objects

This example demonstrates adding a new event field to **/config/adjustment/event** using Developer Center.

1. Start Developer Center.
2. Open the Object Browser.
3. Enter or choose **/config/adjustment/event** in the **Objects** field, then click **Browse**.

   The **/config/adjustment/event** object is displayed in the **Objects** area and the contents of the object are displayed in the Output Flist area.
4. Select and copy the last element in the PIN_FLD_EVENTS array, located at the end of the output flist. It should be similar to the following:

   ```
   0 PIN_FLD_EVENTS     ARRAY [9] allocated 20, used 1
   1  PIN_FLD_TYPE_STR  STR [0] "/event/session"
   ```
5. Switch to Opcode Workbench.
6. Enter or choose PCM_OP_WRITE_FIELDS in the **Opcode** field.
7. Enter **32** in the **Flags** field to specify the PCM_OPFLG_ADD_ENTRY flag. This flag makes it possible to add a new element to the array.
8. In the **Input Flist** area, enter the first line of the input flist as shown below. If necessary, modify the database number.

   ```
   0 PIN_FLD_POID       POID [0] 0.0.0.1 /config/adjustment/event 301
   ```
9. Paste the text you copied in Object Browser onto the next line. The result should be similar to the following:

   ```
   0 PIN_FLD_POID       POID [0] 0.0.0.1 /config/adjustment/event 301
   0 PIN_FLD_EVENTS     ARRAY [9] allocated 20, used 1
   1  PIN_FLD_TYPE_STR  STR [0] "/event/session"
   ```
10. Increment the element number by one. For example, if the element number you pasted is **[9]**, change it to **[10]**.
11. Change the event storable class name in the PIN_FLD_TYPE_STR field to that of the event type you are adding. For example, if the event type you pasted was **/event/session**, you could change it to **/event/session/call/telephony**.
12. Click **Run**.

    The **Output Flist** area displays the object POID to confirm that the opcode ran successfully.
13. Switch to Object Browser, then repeat step 3 to confirm that the new event type has been added.
14. Stop and restart the CM.

Using testnap to Modify /config Objects

This example demonstrates replacing an event field in **/config/adjust/event** using **testnap**. See "Using the testnap Utility to Test BRM" for general instructions and examples for **testnap**.

1. Start **testnap**.

   ```
   % testnap
   ```

2. Use the **robj** command to view the contents of the **/config** object to modify.

   This **robj** command reads the contents of the **/config/adjust/event** object.

   ```
   robj - 0.0.0.1 /config/adjustment/event 301
   ```

3. When the object is displayed, note the element ID of the field to replace in the PIN_FLD_EVENTS array.

4. Create a text file that contains an flist with the field to add. The field must contain the complete POID of the **/config** object and the element you are adding. Set the element ID to the number of the element you are replacing.

   For example, this flist replaces element [5] in the PIN_FLD_EVENTS array in **/config/adjustment/event**.

   ```
   0 PIN_FLD_POID     POID [0] 0.0.0.1 /config/adjustment/event 301 0
   0 PIN_FLD_EVENTS   ARRAY [5] allocated 20, used 1
   1     PIN_FLD_TYPE_STR        STR [0] "/event/session"
   ```

5. Save this file.

6. Read the file into buffer 1. In this example, the file is called **config**.

   ```
   r config1 1
   ```

7. Display the contents of the buffer to verify that it is correct.

   ```
   d 1
   ```

8. Write the contents of the buffer to the **/config** object.

   ```
   wflds 1
   ```

9. Read the object again to verify that your change was made.

   ```
   robj - 0.0.0.1 /config/adjustment/event 301
   ```

10. Stop and restart the CM.

16

Storing Customer Profile Information

Learn how to customize information stored in the Oracle Communications Billing and Revenue Management (BRM) database.

Topics in this document:

- About Storing Customer Profile Information
- Using Profile Objects to Collect Customer Profiles

About Storing Customer Profile Information

You can collect information about your customers and store them in the database in the **/profile** objects. In the profile object, you store marketing or other information relevant to your company, but not necessarily used for accounting. A **/profile** storable class is a top-level class in BRM, which you subclass to define **/profile** storable classes for your specific needs. It contains the standard top-level object fields and a field which is a pointer to the POID of the **/account** object with which it is associated.

The **/account** object is not used to store this marketing information because inheriting the **/account** object to store these fields would cause object-type collisions if a variety of enhanced services were installed on the same BRM installation.

Because **/profile** objects are linked to specific **/account** objects, any number of different **/profile** objects can be linked to the same **/account** object. However, each **/profile** object can be linked to only one **/account** object.

After you define a **/profile** storable class, you use the Customer FM standard opcodes and Customer FM policy opcodes to create, delete, modify, and validate profile objects.

Using Profile Objects to Collect Customer Profiles

This section describes how to use profile objects to collect customer profiles.

Defining a Profile Subclass

Use the Storable Class Editor to create **/profile** storable subclasses.

See "Creating Custom Fields and Storable Classes" for details on adding storable subclasses and fields to the database.

Creating a Profile Object

To create a profile object:

1. Create an flist with a PIN_FLD_INHERITED field containing your specific profile information.
2. Pass this flist into PCM_OP_CUST_CREATE_PROFILE.

Modifying a Profile Object

To modify a profile object:

1. Modify the **/profile** object flist.
2. Pass this flist into PCM_OP_CUST_MODIFY_PROFILE.

Deleting a Profile Object

To delete a **/profile** object, use PCM_OP_CUST_DELETE_PROFILE**.**

Validating Profile Objects

To validate **/profile** objects, you customize the following customer FM policy opcodes:

- PCM_OP_CUST_POL_PREP_PROFILE
- PCM_OP_CUST_POL_VALID_PROFILE

For more information on customizing policy opcodes, see "About System and Policy Opcodes".

17

Auditing Customer Data

Learn how to customize audited data in the Oracle Communications Billing and Revenue Management (BRM) database.

Topics in this document:

- Audit Trail Architecture
- About Shadow Objects
- Fields Marked for Auditing by Default
- Enabling Auditing for a Field
- Accessing Audit Trail Information
- Purging Archived Audit Data

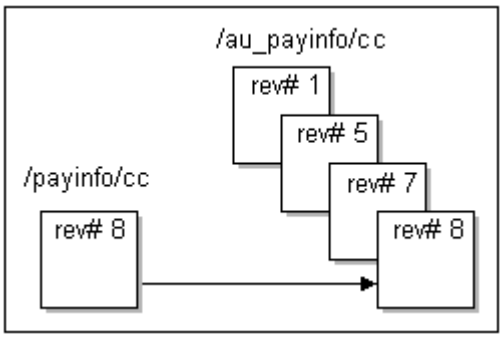
You can enable fields in objects to trigger an audit trail when they are modified. This allows you to track many changes to the BRM system. For information on why you enable an audit trail, see "About Maintaining an Audit Trail of BRM Activity" in *BRM Managing Customers*.

Audit Trail Architecture

You enable audit trails for an individual field, but the entire object that contains the field is versioned and stored to ensure consistency with other members of the storable class. When fields marked for auditing are created or modified, copies are made of the top-level storable class to which the field belongs and of that object's subclasses. The audit trail is composed of the versioned and stored object copies, which are called *shadow objects.* It is created at the end of a transaction, prior to transaction commit.

For example, if a field marked for auditing in the **/payinfo/cc** object is modified in the first, fifth, seventh, and eighth revision of the original object, the shadow objects shown in Figure 17-1 are created in the audit trail:

Figure 17-1 /payinfo/cc Shadow Objects



The revision number is the revision number of the object's Portal object ID (POID).

When you mark a field for auditing, the Storable Class Editor calls the PCM_OP_SDK_SET_DD opcode. This opcode creates a shadow class for the top-level storable class in which the auditable field is modified and for all of that storable class's subclasses.

The auditability of a field is specified in the PIN_FLD_AUDITABLE meta-level field in the data dictionary. The value **1** (**AUDIT_ENABLED**) indicates the field is marked for auditing, and the value **0** (**AUDIT_DISABLED**) indicates the field is not marked for auditing.

The following example shows the input flist of the PCM_OP_SDK_SET_DD opcode after the PIN_FLD_DEBIT_NUM field is marked for auditing in the **/payinfo/cc** storable class. The PIN_FLD_AUDITABLE meta-level field is set to **1**, which indicates that the PIN_FLD_DEBIT_NUM field is marked for auditing.

```
0 PIN_FLD_OBJ_DESC      ARRAY [133] allocated 20, used 4
1     PIN_FLD_NAME            STR [0] "/payinfo/cc"
1     PIN_FLD_DESCR           STR [0] "Credit Card payment
                                       information class."
1     PIN_FLD_OBJ_ELEM      ARRAY [0] allocated 20, used 14
2         PIN_FLD_FIELD_TYPE      INT [0] 9
2         PIN_FLD_FIELD_NAME      STR [0] "PIN_FLD_CC_INFO"
2         PIN_FLD_DESCR           STR [0] "Array to hold the
                                           credit card specific
                                           information. There
                                           can be only one
                                           array element. The
                                           array element id is not
                                           significant."
2         PIN_FLD_ORDER           NUM [0] 0.000000
2         PIN_FLD_OBJ_ELEM      ARRAY [0] allocated 20, used 8
3             PIN_FLD_FIELD_TYPE      INT [0] 5
3             PIN_FLD_FIELD_NAME      STR [0] "PIN_FLD_ADDRESS"
...
...
2         PIN_FLD_OBJ_ELEM      ARRAY [4] allocated 20, used 8
3             PIN_FLD_FIELD_TYPE      INT [0] 5
3             PIN_FLD_FIELD_NAME      STR [0] "PIN_FLD_DEBIT_NUM"
3             PIN_FLD_DESCR           STR [0] "Credit card
                                               number."
3             PIN_FLD_ORDER           NUM [0] 0.000000
3             PIN_FLD_LENGTH          INT [0] 30
3             PIN_FLD_AUDITABLE    INT [0] 1
3             PIN_FLD_CREATE_PERMISSION    STR [0] "Required"
3             PIN_FLD_MOD_PERMISSION    STR [0] "Writeable"
3             PIN_FLD_SM_ITEM_NAME    STR [0] "debit_num"
...
...
```

After a shadow class is created, it is not deleted even if auditing is later disabled. For example, if you disable auditing for the PIN_FLD_DEBIT_NUM field in the **/payinfo/cc** object, the existing audit trail for that field is retained and accessible to you if you need it.

About Shadow Objects

Shadow objects use an **au** prefix. For example, a change to a field marked for auditing in the **/payinfo/cc** object results in the following shadow objects: **/au_payinfo**, **/au_payinfo/cc**, **/au_payinfo/inv**, **/au_payinfo/subord**, and so on. The shadow object is a replica of the original object, so it stores the POID the object had when auditing occurred. The unique revision number of the POID is used to extract audit trail information from the database.

The shadow object contains the same fields as the original object plus the PIN_FLD_AU_PARENT_OBJ field that captures information about the audited storable class. The PIN_FLD_AU_PARENT_OBJ field is a pointer to the revision of the original object copied to the audit trail; its value is derived from the original object field.

You specify the tablespace for the audit trail schema when you install BRM. The default tablespace for a shadow object is the same as that of its original object.

The tablespace name for a shadow object is the same as that of its original object with the addition of an **au** prefix. There is a 32-character limit on tablespace names. If the addition of the **au** prefix exceeds the 32-character limit, the shadow object tablespace name is truncated, causing its tablespace name to be different than that of the original object.

Fields Marked for Auditing by Default

Table 17-1 shows the fields that are marked for auditing by default in BRM, the top-level storable class associated with those fields, the event objects created when those fields are modified, and the BRM activity being audited:

Table 17-1 Fields Marked by Default for Auditing

Top-Level Storable Class	Audited Fields	Event Object Created	BRM Activity Audited
/deal	All fields	**/event/audit/price/deal**	Changes to internal BRM pricing components
/payinfo	PIN_FLD_DEBIT_NUM and PIN_FLD_DEBIT_EXP	**/event/audit/customer/payinfo/cc**	Changes to customer credit card numbers and expiration dates
/plan	All fields	**/event/audit/price/plan**	Changes to internal BRM pricing components
/product	All fields	**/event/audit/price/product**	Changes to internal BRM pricing components
/rate	All fields	**/event/audit/price/rate**	Changes to internal BRM pricing components
/rate_plan	All fields	**/event/audit/price/rate_plan**	Changes to internal BRM pricing components
/ rate_plan_selecto r	All fields	**/event/audit/price/ rate_plan_selector**	Changes to internal BRM pricing components

Enabling Auditing for a Field

You enable and disable auditing for fields using the Storable Class Editor.

Performance tips:

- Keep an audit trail only for the BRM activity that is absolutely necessary for your business. Enabling an audit trail decreases system performance significantly.
- Do not mark fields in the **/account** object for auditing. Because the **/account** object is large and is modified by many types of system activity, it is not recommended for auditing. Mark the **/purchased_product** and **/purchased_discount** objects for auditing to audit charge offers and discount offers for an account.

- If you mark an array or substruct for auditing, changes to any field in that array or substruct trigger auditing. To track changes to only one field in an array or substruct, mark *only* that field for auditing and *not* the entire array or substruct.

Accessing Audit Trail Information

You can access audit trail information manually by using the **testnap** utility to retrieve specific revisions of an object given the object's POID. You can also write your own application to access audit trail changes in the database.

To access audit trail data, you must:

- Obtain the account number of the customer for whom you need audit trail data (as shown in Billing Care).
- Obtain the general time period in which the event that was audited occurred. For example, the date or month the customer changed his credit card number.
- Know the name of the event object BRM generates for each type of auditing. For example, when a customer changes the credit card number on his account, BRM generates the event object **/event/audit/customer/payinfo/cc**.

  For the names of the event objects generated for default auditing, see "Fields Marked for Auditing by Default".

1. Use the customer's account number to obtain the **/account** object associated with the account number.
2. Browse the event objects for the **/account** object in the general time period to locate the event. For example, to find a changed credit card number, browse the events to locate the **/event/audit/customer/payinfo/cc** event created at the time the customer changed his credit card number.
3. Use the event object to obtain the POID of the original object that was audited. For example, the **/event/audit/customer/payinfo/cc** event contains the POID of the **/payinfo/cc** object that was audited when the customer's credit card number was changed. The POID provides the revision number of the object when it was audited. This number is stored in the shadow object.
4. Retrieve the shadow objects by calling the PCM_OP_READ_OBJ opcode with one of the following flags. (For examples of how to do this by using the **testnap** utility, see "Using testnap to Retrieve Shadow Objects".)
   - Use the PCM_OPFLG_USE_POID_GIVEN flag to send a request to the BRM database Data Manager (DM) to run a search-and-read operation for the exact POID you specify (the POID of the shadow object in the audit trail).
   - Use the PCM_OPFLG_USE_POID_PREV flag to send a request to the BRM database DM to run a search-and-read operation for the shadow object POID that contains a revision number preceding the revision number you specify in your POID.
   - Use the PCM_OPFLG_USE_POID_NEXT flag to send a request to the BRM database DM to run a search-and-read operation for the shadow object POID that contains a revision number that is one or more numbers higher than the revision number you specify in your POID.
   - Use the PCM_OPFLG_USE_POID_NEAREST flag to send a request to the BRM database DM to run a search-and-read operation for the exact audit trail revision number that you specify (the POID of the shadow object), and if the exact POID is not found, to obtain the POID with a revision number that precedes the revision number you specify in your POID.

Using testnap to Retrieve Shadow Objects

The following code samples show how to retrieve shadow objects by using **testnap**. This example accesses the audit trail of the **/payinfo/cc** object given the object's POID near the time auditing occurred.

```
# Flists and opcode for testnap to access the audit-trail
# of the /payinfo/cc object.
#
# Syntax: xop PCM_OP_READ_OBJ flags buffer
#
# where the buffer contains the following flist with the
# flags as described below:
#
#
0 PIN_FLD_POID POID [0] 0.0.0.1 /payinfo 10001 0


# NOTE: Replace the database number, the POID_ID, and
# the version as required.

#Use the following flags:

# PCM_OPFLG_USE_POID_GIVEN (0x0040): To access the exact revision.
# e.g. "xop PCM_OP_READ_OBJ 0x0040 1" on the buffer flist
# 0 PIN_FLD_POID POID [0] 0.0.0.1 /payinfo 10001 3
# This retrieves audit trail with revision 3, or NOT_FOUND.

# PCM_OPFLG_USE_POID_PREV (0x2000): To access the
# previous revision.
# e.g. "xop PCM_OP_READ_OBJ 0x2000 1" on the buffer flist
# 0 PIN_FLD_POID POID [0] 0.0.0.1 /payinfo 10001 3
# This retrieves audit trail with revision 2, assuming revision
# 2 is the previous revision to 3 in the audit-trail. Otherwise,
# the next revision lower than revision 3 is retrieved.

# PCM_OPFLG_USE_POID_NEXT (0x4000+): To access the next revision.
# e.g. "xop PCM_OP_READ_OBJ 0x4000 1" on the buffer flist
# 0 PIN_FLD_POID POID [0] 0.0.0.1 /payinfo 10001 3
# This retrieves audit trail with revision 4, assuming revision
# 4 is the next revision to 3 in the audit trail. Otherwise, the
# next revision higher than revision 3 is retrieved.


# PCM_OPFLG_USE_POID_NEAREST (0x8000): To access the nearest
# revision.
# e.g. "xop PCM_OP_READ_OBJ 0x8000 1" on the buffer flist
# 0 PIN_FLD_POID POID [0] 0.0.0.1 /payinfo 10001 3
# This retrieves audit trail with revision 3, if the revision 3
# exists; otherwise, it retrieves a revision preceding
# revision 3.
```

Purging Archived Audit Data

Use the **purge_audit_tables.pl** Perl script to remove unwanted audit data from your audit tables by moving older rows to history (archive) tables. Purging the audit tables improves system performance and reduces memory usage,

> **Note:**
>
> The **purge_audit_tables.pl** script does not delete objects from the database; it only purges the object rows stored in a table.

To purge objects from audit tables:

1. Open the *BRM_home*/**sys/archive/oracle/purge_audit_tables.conf** file.
   a. In the **storage_clause** entry, specify the tablespace for the history tables.
   b. In the **time** entry, specify the column name to be used for comparing the cutoff date specified in the **purge_audit_tables.pl** script's **-d** parameter.
   c. In the **cutoff_for_purge** entry, specify the percentage based on which it will invoke the **archiveindirect** mode rather than the **archivedirect** method to archive the tables.

      For example, if the **cutoff_for_purge** value is **70**, and a table contains more then 70% data that must be archived, temporary tables are used to transfer the data efficiently (**archiveindirect** mode). If the table contains less then 70% data that must be archived, the data is transferred directly to the history tables (**archivedirect** mode).

   For more information about the configuration entries, see the **purge_audit_tables.conf** file in the *BRM_home*/**sys/archive/oracle** directory.
2. With a text editor, open the **purge_audit_tables.pl** script.
3. In the first line of the script, replace __**PERL**__ with the location of the Perl executable.
4. Run the **purge_audit_tables.pl** script. See "purge_audit_tables.pl" in *BRM System Administrator's Guide* for more information.

> **Note:**
>
> To run in debug mode, set the environment variable ARCHIVE_DEBUG at the system prompt before you run the script. As the script runs, processing data, including the functions that are called, is printed to the screen.

18

Encrypting Data

Learn how to encrypt data in the Oracle Communications Billing and Revenue Management (BRM) database.

Topics in this document:

- About Encrypting Data
- About AES Encryption
- About Masking Data in Log Files
- Encrypting Fields
- Defining Masked Fields
- About Encrypting Passwords
- Configuring the Data Manager for Oracle ZT PKI Encryption
- Configuring the Data Manager for AES Encryption
- Generating a Root Encryption Key
- Modifying a Root Encryption Key

About Encrypting Data

You can encrypt fields that contain sensitive customer information, such as credit card numbers, to guarantee privacy and prevent unauthorized use. The fields to be encrypted must be in string format. You set up encryption with the Storable Class Editor, which will add a flag attribute in the metadata defining the field in the data dictionary (PIN_FLD_ENCRYPTABLE).

BRM encrypts the fields marked for encryption when storing them in the database and automatically decrypts the fields when retrieving them from the database. For information on how to encrypt fields, see "Encrypting Fields".

You can also encrypt passwords, including the database password and passwords for servers and client applications that connect to the Connection Manager (CM) to access the database. For information, see "About Encrypting Passwords".

About AES Encryption

AES uses a 256-bit encrypted key to protect field data. To set up the database for AES encryption, you:

- Generate an encrypted AES key and add it to the Data Manager (DM) **pin.conf** file with other encryption configuration information. You can have only one encrypted AES key per Oracle DM.

  The encrypted AES key gets stored in the database.
- Define which fields should be encrypted by setting their PIN_FLD_ENCRYPTABLE flag in Storable Class Editor.

When the Oracle DM starts, it uses the encrypted AES key to transform the fields marked as encryptable from plaintext into ciphertext and from ciphertext into plaintext.

In addition to encrypting data fields, you can use AES to encrypt passwords for these features:

- BRM Data Managers
- Pipeline database
- Optional managers, such as Account Synchronization Manager
- Client applications

About AES Data Encryption Length

The length of AES-encrypted data is different depending on whether the data is stored in the database or not. In both cases, encrypting a field increases its length.

> **Note:**
>
> The maximum length of an encrypted field in the Oracle database is 1,992 bytes, which is 975 bytes in plaintext.

Encryption Length for Fields Stored in the Database

When the AES encryption scheme is used to store database fields, the generated ciphertext is in this format:

&aes|*Encrypted_AES_key_index***|***Ciphertext*

where:

- **aes** identifies the encryption scheme used by the database.
- *Encrypted_AES_key_index* is a unique 4-digit ID associated with the encrypted AES key.
- *Ciphertext* is the encrypted data generated from the plaintext data by using the AES key.

The length of the generated ciphertext data is:

(*Plaintext_data_length* **+ 16**) *** 2 + 5 + 5**

Two hexadecimal numbers represent each byte in the ciphertext, **&aes |** is equal to 5 bytes, and *Encrypted_AES_key_index* is equal to 5 bytes. For example, if *Plaintext_data_length* is 25, the length of the generated ciphertext data is **92**:

(25+16)*2+5+5 = 92

> **Note:**
>
> If *Plaintext_data_length* is less than 17, the length of the generated ciphertext data is always **79**.

Encryption Length for Fields Not Stored in the Database

When the AES encryption scheme is used to encrypt data that is not stored in the database, such as passwords, the generated ciphertext is in this format:

&aes|*Ciphertext*

where:

- **aes** identifies the encryption scheme used by the database.
- *Ciphertext* is the encrypted data generated from the plaintext data by using the AES key.

The length of the generated ciphertext data is:

(*Plaintext_data_length* + **16**) *** 2 + 5**

A hexadecimal number represents each byte in the ciphertext, and **&aes |** is equal to 5 bytes. For example, if *Plaintext_data_length* is 25, the generated ciphertext data is **87**:

(25+16)*2+5 = 87

> **Note:**
>
> If *Plaintext_data_length* is less than 17, the length of the generated ciphertext data is always **74**.

Generating an Encrypted AES Key

An encrypted AES key is used by the DM to encrypt the database fields that are marked as encryptable.

To generate an encrypted AES key:

1. Run the **pin_crypt_app** utility:

   ```
   pin_crypt_app -genkey -key AES_key
   ```

   If you do not have an AES key, run the utility with only the **-genkey** parameter. This generates a random AES key internally and then encrypts it with a hidden key to create the encrypted AES key. The key length must be 64 and can contain only hexadecimal numbers.

   The output states whether the key was generated successfully, and if so, provides the encrypted AES key.
2. Write down the encrypted AES key value or copy it to a text editor. Include the &aes| because it is part of the encrypted key.
3. Add the encrypted AES key value to the crypt entry in the DM **pin.conf** file.

You can have only one encrypted AES key per Oracle DM.

Replacing an Encrypted AES Key

You can replace an encrypted AES key with a new key at any time. This does not affect how data is decrypted; the DM can decrypt data encrypted with a previous key.

This procedure assumes you have already configured the DM for AES encryption. See "Configuring the Data Manager for AES Encryption".

To replace an encrypted AES key:

1. Generate a new encrypted AES key. See "Generating an Encrypted AES Key".
2. Open the DM **pin.conf** file.

3. In the **crypt aes** entry, replace the existing encrypted AES key with the new encrypted AES key:

   ```
   - crypt aes|BRM_home/lib/libpin_crypt_aes4dm.so "&aes|New_encrypted_aes_key"
   ```

4. Save the file.
5. Stop and restart the DM.

About Masking Data in Log Files

Currently, fields defined as encryptable are encrypted by the DM when they are stored in the database and decrypted when they are retrieved. Data passed through the CM by opcodes and data in the client applications is in plaintext. Therefore, when BRM opcodes are called and high log levels are set on flist operations, the contents of the encryptable fields are saved to a log file. The fields may contain sensitive data that is defined as encryptable but still appears in plaintext. To hide this information, you can define a field as masked. The masked data will be displayed as **"XXXX**" in the log files rather than as plaintext.

For information on how to mask encrypted fields during flist logging, see "Defining Masked Fields".

Encrypting Fields

You mark fields for encryption by using the Storable Class Editor. You can enable new or existing object fields for encryption at any time.

> **Note:**
>
> - You can disable encryption for a field at any time; however, it is recommended that you only do so during upgrades.
> - Make sure the field is in string format. Only strings may be encrypted.

Defining Masked Fields

You can mask BRM fields and custom fields.

To define masked fields:

1. Create a custom file for your masked fields, and define the fields in this file. Use the following syntax:

   ```
   Custom_field_name masked
   ```

   For example, you might create a file named **custom_field_attributes** that contains the following mask definitions:

   ```
   CUST_FLD_CC_NUMBER masked
   CUST_FLD_EXPIRY_DATE masked
   ```

2. Generate the source file for your custom fields. By default, this file is called **custom_fields.h**.
3. Copy the contents of both files (the custom masked file and the source file) to a new file. For example, name the new file **custom_masked_fields**.

4. Run the **parse_custom_ops_fields** Perl script, and use the custom masked fields file as the *input*:

   ```
   parse_custom_ops_fields -L language -I input -O output -P java_package
   ```

5. For applications written with PCM C, add the following entry to the **pin.conf** file for each PCM C application:

   ```
   - - ops_fields_extension_file ops_flds_ext
   ```

6. For applications written using Java PCM, including Developer Center, copy the contents of the **InfranetPropertiesAdditions.properties** file and paste it into the **Infranet.properties** files for each Java application.

For more information about field masking, see "About Masking Data in Log Files".

About Encrypting Passwords

You can encrypt passwords for the BRM database, optional managers such as GSM Manager, server applications, and client applications that use a password to connect to the CM.

These passwords can be encrypted manually (one at a time) by running the **pin_crypt_app** utility or automatically (all at one time) by running the **encryptpassword.pl** script.

> **Note:**
>
> You must use the **pin_crypt_app** utility to encrypt client application passwords or passwords associated with customizations (for example, custom passwords in BRM-provided configuration files or passwords in non-BRM configuration files that support custom applications).

For information on the **encryptpassword.pl** script, see "About the encryptpassword.pl Script".

For information about configuring client applications to connect to the CM without a password, see "Using CM Proxy to Allow Unauthenticated Log On" in *BRM System Administrator's Guide*.

To customize password encryption, use the PCM_OP_CUST_POL_ENCRYPT_PASSWD, PCM_OP_CUST_POL_COMPARE_PASSWD, and PCM_OP_CUST_POL_DECRYPT_PASSWD opcodes. See *BRM Opcode Guide*.

> **Note:**
>
> When you change a password, it is not automatically encrypted. You must encrypt the new password and update the entry in the appropriate configuration file.

About the encryptpassword.pl Script

You run the **encryptpassword.pl** script to encrypt the passwords for all BRM components at one time, including the Oracle DM password. BRM authenticates these passwords before connecting to the CM or the BRM database.

> **Note:**
>
> This script does not encrypt passwords for client applications or optional managers that are not part of base BRM. In addition, it does not encrypt passwords associated with customizations; for example, custom passwords in BRM-provided configuration files or passwords in non-BRM configuration files that support custom applications. To encrypt such passwords, run the **pin_crypt_app** utility.

The **encryptpassword.pl** script has no parameters. You run it from the Linux prompt on the system running the BRM database by entering the following command:

```
perl encryptpassword.pl
```

This script performs the following tasks on the machine on which it runs:

1. Creates a backup copy of all **pin.conf** and **Infranet.properties** configuration files in which it finds a password.
2. Replaces the plaintext password in each configuration file with an encrypted password.
3. Adds all encrypted passwords to the **pin_setup.values** file.
4. Adds the ENABLE_ENCRYPTION entry with a **YES** value to the **pin_setup.values** file. This field enables password encryption.

Encrypting Passwords Automatically for BRM Base Components

To encrypt passwords for all BRM base components at one time:

1. Log into the system running the BRM database.
2. Go to the *BRM_home***/setup/scripts** directory.
3. Run the **encryptpassword.pl** script:

   ```
   perl encryptpassword.pl
   ```

4. Follow the instructions at each prompt.

Passwords are encrypted in the OZT format.

> **Note:**
>
> - If you are running optional managers or server applications on a system that does not contain the BRM database, run the **encryptpassword.pl** script on each applicable system.
> - If you change an encrypted password, you must update the entry in the configuration file and the **pin_setup.values** file.

For more information on the **encryptpassword.pl** script, see "About the encryptpassword.pl Script".

Encrypting Passwords Manually with OZT

To encrypt passwords manually with OZT:

1. Log in to the system running the BRM manager.

   > **Note:**
   >
   > For data managers, this is generally the system running the BRM database; for client applications, it is the application host system.

2. Run the **pin_crypt_app** utility with the **-useZT** parameter:

   ```
   pin_crypt_app -useZT -enc
   ```

3. At the prompt, enter the plaintext password to encrypt and then re-enter it again.

   The output is the OZT-encrypted password.

4. Write down the encrypted password or copy it to a text editor.

> **Note:**
>
> When you change a password, it is not automatically encrypted. You must encrypt the new password and update the entry in the appropriate configuration file.

To set this password as the Oracle DM password or an optional manager password, add the password to the manager's **pin.conf** file. See "Configuring the Data Manager for Oracle ZT PKI Encryption".

To set this password as the CM password for a client application, add the password to the application's **pin.conf** file and the **Infranet.properties** file. By default, the **Infranet.properties** file is located in **C:\Program Files\Common Files\Portal Software**.

Encrypting Passwords Manually with AES

To encrypt passwords manually with AES:

1. Log in to the system running the BRM manager.

   > **Note:**
   >
   > For data managers, this is generally the system running the BRM database. For client applications, it is the application host system.

2. Run this command to transform a plaintext password into ciphertext:

   ```
   pin_crypt_app -enc
   ```

3. At the **Enter the plaintext to encrypt** prompt, enter your plaintext password.
4. Re-enter your plaintext password.

The output is the AES-encrypted password.

5. Write down the encrypted password or copy it to a text editor.

To set this password as the Oracle DM password or an optional manager password, add the password to the manager's **pin.conf** file. See "Configuring the Data Manager for AES Encryption".

To set this password as the Pipeline Manager password, add the password to the **DataPool** section of the Pipeline startup registry file.

To set this password as the CM password for a client application, add the password to the application's **pin.conf** file and the **Infranet.properties** file.

Configuring the Data Manager for AES Encryption

The Oracle DM configuration file (**pin.conf**) specifies the user name and password needed to log in to the BRM database as well as the encryption method to use for data stored in the BRM database.

To configure the Oracle DM for AES encryption:

1. If necessary, generate an encrypted AES key and an AES encrypted password. See "Generating an Encrypted AES Key" and "Encrypting Passwords Manually with AES".
2. Open the *BRM_home*/**sys/dm_oracle/pin.conf** file in a text editor.
3. Add the AES encrypted password for logging in to your BRM database to the **sm_pw** entry:

   ```
   - dm sm_pw Encrypted_password
   ```

   The password can use any character from the US7ASCII character set except for the hash character (#). BRM interprets the hash character as a comment and will ignore any subsequent characters in the password.
4. Set AES as the encryption method for your data by using the **crypt** entry:

   ```
   - crypt aes|Encryption_library "&aes|Encrypted_aes_key"
   ```

   where:

   - *Encryption_library* is the path and filename of the AES encryption library (**pin_crypt_aes4dm**). The prefix for the library is **lib** for Linux, or **null ""** for Windows. The extension for the library is **.so** for Linux, and **.dll** for Windows.
   - *Encrypted_aes_key* is your encrypted AES key. See "Generating an Encrypted AES Key".
5. Save the file.
6. Stop and restart the DM.

Configuring the Data Manager for Oracle ZT PKI Encryption

The Oracle ZT public key infrastructure (PKI) encryption algorithm uses the **pin_crypt_app** and **pin_config_editor** utilities to encrypt files, plain text, passwords, and root keys.

The Oracle DM configuration file (**pin.conf**) specifies encryption settings for the database.

To configure the Oracle DM to use OZT encryption:

1. If necessary, generate an OZT encrypted password. See "Encrypting Passwords Manually with OZT".
2. Run the following command to generate an encrypted Oracle ZT PKI key:

   ```
   pin_crypt_app -useZT genkey [-key Key]
   ```

   where *Key* is a 256-bit key in hexadecimal notation.
3. Write down the encrypted Oracle ZT PKI key value, or copy it to a text editor.

   > **Note:**
   >
   > Include **&ozt|** because it is part of the encrypted Oracle ZT PKI key value.

4. Go to *BRM_home*/**sys/dm_oracle**.
5. Open the Oracle DM **pin.conf** file in a text editor.
6. Enable the Oracle ZT PKI encryption algorithm by adding the following entry:

   ```
   - crypt aes|Encryption_library "&ozt|Encrypted_key"
   ```

   where

   - *Encryption_library* is the path and filename of the OZT encryption library (**pin_crypt_aes4dm**). The prefix for the library is **lib** for Linux, or **null ""** for Windows. The extension for the library is **.so** for Linux, and **.dll** for Windows.
   - *Encrypted_key* is the OZT encrypted key that you generated in step 2.
7. Save the file.
8. Stop and restart the Oracle DM.

Generating a Root Encryption Key

The root encryption key is used for encrypting the data and password using the Oracle ZT PKI–approved encryption algorithm.

To generate a root encryption key:

1. Go to *BRM_home*/**bin**.
2. Run the following command, which creates a root key wallet:

   ```
   orapki wallet create -wallet wallet_location -pwd password
   ```

   where:

   - *wallet_location* is the directory in which the root key wallet is to be created.
   - *password* is the password used to make changes to the root key wallet.
3. Run the following command:

   ```
   pin_crypt_app -genrootkey
   ```

   A root encryption key is generated and is stored in the root key wallet.

> **Note:**
>
> The root key wallet is generated at only one location. You can copy the root key wallet to the other locations.

Modifying a Root Encryption Key

To enhance security, you should modify the root encryption key on a regular basis. For detailed instructions, see "Modifying the Root Encryption Key" in *BRM Installation Guide*.

19

Searching for Objects in the BRM Database

Learn about the Oracle Communications Billing and Revenue Management (BRM) object search strategy, including the types of searching that BRM performs by default and what you must know about searching if you are writing custom applications to use with BRM.

Topics in this document:

- About Searching for Objects
- About the Search Input Flist
- Search Query Syntax
- Searching Subclasses
- Returning Specific Storable Classes
- Returning Entire Arrays
- Search Template Examples
- About Single-Schema Searches
- Performing Exact Searches
- Complex Searches
- Search without POID
- About Multischema (Global) Searches
- The Impact of Searches on Shared Memory Allocation
- Improving Search Performance

About Searching for Objects

Searching in this context means looking in the BRM database for objects that meet your specified criteria. You search for the Portal object IDs (POIDs) of all the storable classes with specific characteristics.

> **Note:**
>
> If you know the POID of the storable class you are searching for, you can use PCM_OP_READ_OBJ and PCM_OP_WRITE_OBJ to read and change data.

There are two main types of searching:

- Simple searching on a single storable class and its inherited classes. This usually means searching for a specific account.
- Complex searching across multiple storable classes at the same time. For example, searching for all accounts in a specific city that used a particular service includes both the **/account** and **/service** storable classes.

Depending on your implementation, these searches can be performed on one or more database schemas.

There are two other options for the SEARCH opcodes:

- Count-only searches count and return the number of POIDS that match your search criteria. They do not return POIDs. Use the PCM_OPFLG_COUNT_ONLY parameter to perform a count-only search.
- Calculate-only searches return a single calculated value, such as a sum or average. Use the PCM_OP_CALC_ONLY_1 parameter to perform a calculate-only search.

To search for objects in the database, you use a *search template*. The template can be predefined and stored in a **/search** object or defined at runtime when a search opcode is called. When you define the template at runtime, you include the search query on the search opcode input flist. The advantage of defining the template at runtime is that you do not have to create and store it in the database first.

BRM includes predefined **/search** objects in *BRM_home***/sys/dd/data/init_objects.source** that you can use as templates. (*BRM_home* is the directory in which you installed the BRM server software.) The predefined search templates are stored in the SEARCH_T database table. Each template has a predefined ID, such as **230** or **231**.

You can also create your own search objects. Before defining a search, look in the **init_objects.source** file to see whether a template for your search exists. If not, create one and load it into the database. Your **/search** object can then be used in a call to the search opcode.

If you create a predefined search template, add a copy of it to **init_objects.source**. Adding the template has the following advantages:

- You avoid assigning duplicate search template IDs.
- All search templates are in a single location, making them easy to find.
- In the testing mode, you can automatically load the new search templates if you must re-create the database.

> **Note:**
>
> When upgrading to a new BRM release, ensure you copy your custom templates to the new **init_objects.source** file.

About the Search Input Flist

You specify the search criteria and the results you want to be returned from the search on the input flist of the search opcode.

The input flist requires three fields:

- **Search POID**: Specifies a search object. See "Search POID".
- **PIN_FLD_ARGS array**: Specifies the arguments in the search criteria. See "Argument List".
- **PIN_FLD_RESULTS array**: Specifies which fields to return from the search. See "Results Array".

When you define a search template at runtime (when the search is run), two more fields are required:

- **PIN_FLD_TEMPLATE**: Specifies the search query in the form of a string. See "Search Query".
- **PIN_FLD_FLAGS**: Specifies the type of search to perform. See "Flags".

An optional PIN_FLD_PARAMETERS field can be included in the flist when you use a predefined search template. This field specifies a subclass that contains the search arguments. See "Using the PIN_FLD_PARAMETERS Field".

You can use the optional PIN_FLD_MIN_ROW and PIN_FLD_MAX_ROW input flist fields to specify the start and end rows of the search result to retrieve from the search result set. See "Limiting Search Results by Using Row Numbers".

Search POID

The search POID identifies the search template to use for the search.

- If you predefine a search template and store it in a search object, add the POID of the **/search** object to the input flist. The POID specifies which predefined search template to use:

```
0 PIN_FLD_POID          POID [0] 0.0.0.1 /search 301 0
```

- If you define the search template at runtime, add the search object POID to the input flist with an object ID of **0** or **-1**, and define the search query in a PIN_FLD_TEMPLATE field on the input flist:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_TEMPLATE        STR [0] "select X from /account where F1 like V1"
```

Argument List

You specify the arguments for the search query in the PIN_FLD_ARGS array. Each array element contains one argument. You must provide at least one argument.

Always include the POID of the storable class type you want to be returned in the PIN_FLD_ARGS array. If you do not, the search returns an error.

The maximum number of search arguments is 32. The array element ID specifies which argument is contained in the array element. For example, element-ID 1 corresponds to argument 1.

The arguments in the PIN_FLD_ARGS array are referenced in the **where** clause of the search query. If the search criteria specified in the **where** clause exists in a storable class other than the one specified in the search query, an attempt is made to convert the POID type of the unspecified storable class to the specified storable class, which causes an error.

Results Array

You specify in the opcode's PIN_FLD_RESULTS array which fields to return from the objects you're searching. The opcode returns one element for each matching object.

> **Note:**
>
> You specify the objects themselves in the PIN_FLD_ARGS array.

The search can return results in these ways:

- To return all fields from a matched object, set the PIN_FLD_RESULTS input flist element to NULL.
- To return only a count of matching objects, set the PIN_FLD_RESULTS input flist element to NULL and pass the PCM_OPFLG_COUNT_ONLY flag in the opcode call. The opcode returns the number of matching results as the element ID of the PIN_FLD_RESULTS output flist array.
- To return a specified list of fields, set the PIN_FLD_RESULTS input flist element with the list of fields to return.
- To return a single value that is calculated from the matched objects, put the PIN_FLD_RESULTS element that includes a single PIN_FLD_AMOUNT field on the input flist. Also, set the search flag to SRCH_CALC_ONLY.

To indicate the maximum number of records to return, specify that number as the element ID of the PIN_FLD_RESULTS element. To return all records, use zero. For count-only or calculate-only searches, only one value is returned.

If you call for a calculate-only search using the PIN_FLD_CALC_ONLY_1 flag, the PIN_FLD_RESULTS array must contain only one PIN_FLD_AMOUNT field with an element ID of **1**.

Search Query

When defining a search template at runtime, specify the search query in the PIN_FLD_TEMPLATE field. The template is in the form of an SQL-like search string. For example, "select X from <*object*> where <*expression*>". For more information, see "Search Query Syntax".

Flags

You can specify the following searches using the PIN_FLD_FLAGS field on the input flist:

- **256**: SRCH_DISTINCT
  This search type returns only unique data. Use this option when searches join to other objects or access arrays within a single object such that a query might return multiple copies of the same object. To skip this feature, set the value to **0**.

  When there is no possibility of returning multiple copies of the same object, using the SRCH_DISTINCT flag becomes redundant and can degrade search performance. To prevent this from happening, see "Removing Redundant Distinct Searches".

> **Note:**
>
> This flag performs distinct operations on POIDs only. If any other data type is used in the input query, SRCH_DISTINCT does not return distinct results. This flag cannot be used with an **order by** clause while performing a complex search. In this case, set the flag value to **0**. Alternatively, see "About Performing Distinct Searches with Ordering and Pagination" for another way to perform these searches.

- **512**: SRCH_EXACT
  Use this flag to search arrays. This flag applies the **where** clause in the search string to arrays. If this flag is not used, the search opcode might return array elements that do not match the search criteria. For more information, see "Performing Exact Searches".

  > **Tip:**
  >
  > To specify both SRCH_EXACT and SRCH_DISTINCT flags, add their values (512 and 256) and enter **768** in the PIN_FLD_FLAGS field.

- **1024**: SRCH_WITHOUT_POID
  This search type returns data without POIDs for each result. For more information, see "Search without POID".

There are also two flags that you can use in the call to a search opcode:

- PCM_OPFLG_COUNT_ONLY
  This search type returns only the number of POIDs that match the search criteria. The value of the PIN_FLD_RESULTS element on the input flist must be NULL.
- A calculate-only flag
  This search type returns a value, such as a sum or average. There are two forms of the calculate-only flag:
  - SRCH_CALC_ONLY_1
    Use this flag to return a single value.
  - SRCH_CALC_ONLY
    Use this flag to return one or more values.

  For more information, see "Search Query Syntax for Calculate-Only Searches".

Search Query Syntax

The search query is part of the search template. You include it in the predefined template before storing it in the database or the opcode's PIN_FLD_TEMPLATE input flist field at runtime.

> **Note:**
>
> SQL queries must adhere to the limitations imposed by the database. For information, see your database documentation.

Use the following syntax for search queries:

"select *X* from *object_name* where *expression*"

where:

- *X* is a placeholder for the field(s) being requested, which are specified in the PIN_FLD_RESULTS array on the input flist.
- *object_name* is the type name of the object that contains the argument(s).

  The name can be fully specified; that is, it can include the specific subclass, or it can take an optional parameter (for example, **/event/$1**). The **$1** parameter is substituted with the value of the PIN_FLD_PARAMETERS field on the search flist. If PIN_FLD_PARAMETERS is not included on the flist, the **$1** is null. See "Using the PIN_FLD_PARAMETERS Field".

  > **Note:**
  >
  > The storable class type you specify in the search query tells BRM where to find the arguments in the **where** clause. It does not indicate the storable class type to return.

- *expression* is an SQL expression such as "where **F1** = **V1** and **F2** = **V2**".

  The column names and literal values (**F***n* and **V***n*) are replaced by the field names and field values specified in the PIN_FLD_ARGS array on the input flist. The column name and value indexes must be contiguous and correspond with the elements in the arguments array starting with element ID 1. That is, F1 and V1 correspond to the field name and value in PIN_FLD_ARGS[1], F2 and V2 to the field name and value in PIN_FLD_ARGS[2], and so on.

About Searching for Objects by Their POID Subcomponent

You can search for objects by specifying any of the following POID subcomponents in the search expression:

- Database number
- Storable class type
- Object ID
- Revision number

In addition to the F*n* = V*n* expression, you can use any one of the following expressions in the **where** clause of the search query template:

- **F***n***.db =V***n* to search by the POID database number. See "Searching for Objects by the POID Database Number".
- **F***n***.type = V***n* to search by the POID type. See "Searching for Objects by the POID Type".
- **F***n***.id = V***n* to search by the POID object ID. See "Searching for Objects by the POID Object ID".
- **F***n***.rev = V***n* to search by the POID revision number. See "Searching for Objects by the POID Revision Number".

> **Tip:**
>
> SQL expressions for searching Oracle databases can include optimizer hints. You can use any hint supported by Oracle. See the Oracle documentation for complete information.

Searching for Objects by the POID Database Number

This example shows the PCM_OP_SEARCH input flist with the POID database number specified in the search query:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search/pin 0 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /service where F1.db = V1 "
0 PIN_FLD_RESULTS       ARRAY [*] allocated 20, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/ip -1 0
0 PIN_FLD_ARGS          ARRAY [1] allocated 20, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/% 1 0
```

Searching for Objects by the POID Type

This example shows the PCM_OP_SEARCH input flist with the POID type specified in the search query:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search/pin 0 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /service where F1.type like V1 "
0 PIN_FLD_RESULTS       ARRAY [3] allocated 1, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/ip -1 0
0 PIN_FLD_ARGS          ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/IP 1 0
```

Searching for Objects by the POID Object ID

This example shows the PCM_OP_SEARCH input flist with the POID object ID specified in the search query:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search/pin 0 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /service/ip where F1 like V1 AND F2.id
= V2"
0 PIN_FLD_RESULTS       ARRAY [0] allocated 2, used 2
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/ip -1 0
1   PIN_FLD_ACCOUNT_OBJ  POID [0] 0.0.0.1 /account -1 0
0 PIN_FLD_ARGS          ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /service/ip -1 0
0 PIN_FLD_ARGS          ARRAY [2] allocated 1, used 1
1   PIN_FLD_ACCOUNT_OBJ  POID [0] 0.0.0.1 /account 24295 1 0
```

Searching for Objects by the POID Revision Number

This example shows the PCM_OP_SEARCH input flist with the POID revision number specified in the search query:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search/pin 0 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /account where F1.rev = 10 "
```

```
0 PIN_FLD_RESULTS       ARRAY [0] allocated 2, used 2
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account -1 0
1   PIN_FLD_NAMEINFO    ARRAY [*] allocated 0, used 0
0 PIN_FLD_ARGS          ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account 1897978 0
```

Search Query Syntax for Count-Only Searches

You can search using the following syntax:

"select *result* **from** *object_name* **where** *expression*"

where *result* is a count value returned when a search is performed using the PCM_OPFLG_COUNT_ONLY flag. This flag returns only the number of matches found by the search.

The following example shows the PCM_OP_SEARCH input flist for counting the total journal objects:

```
...
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_TEMPLATE       STR [0] "select X from /journal where F1 = V1 "
0 PIN_FLD_FLAGS          INT [0] 0
0 PIN_FLD_ARGS           ARRAY [1]
1 PIN_FLD_POID           POID [0] 0.0.0.1 /journal -1 0
0 PIN_FLD_RESULTS          ARRAY [*] NULL
...
```

To run opcodes using **testnap**, use the **xop** command:

```
xop PCM_OP_SEARCH 0x10 1
```

where **0x10** is a PCM_OPFLG_COUNT_ONLY flag and **1** is the buffer.

The opcode returns the count as the index of the PIN_FLD_RESULTS array. For example:

```
...
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_RESULTS       ARRAY [7] NULL array ptr
...
```

where **7** is the total journal count.

Search Query Syntax for Calculate-Only Searches

You can also search using this syntax:

"select *result* from *object_name* where *expression*"

where *result* is a calculated value returned when a search is performed by using the calculate-only flag. There are two forms of the calculate-only flag:

- SRCH_CALC_ONLY_1

This flag returns a single value. You specify one calculation in the search query. For example:

"select **sum(F1)** from *object_name* where *expression*"

> **Note:**
>
> You must include spaces around **F1**.

F1 references the field value of PIN_FLD_ARGS array element 1 on the input flist. You must include the PIN_FLD_AMOUNT field with an element ID of **1** as the only field in the PIN_FLD_RESULTS array.

- SRCH_CALC_ONLY

  This flag can return one or more values. You specify each calculation in the search query. For example:

  "select **sum(F1), avg(F2)** from *object_name* where *expression*"

  For multiple results, you must include a PIN_FLD_AMOUNT field in the PIN_FLD_RESULTS array for each result to return.

  If you use SRCH_CALC_ONLY with the PCM_OPFLG_SRCH_CALC_RESULTS flag not set (which is the default), all PN_FLD_AMOUNT values are returned in the PIN_FLD_RESULTS array. For example:

```
...
0 PIN_FLD_RESULTS      ARRAY [0]
1   PIN_FLD_AMOUNT   DECIMAL [1]
1   PIN_FLD_AMOUNT   DECIMAL [2]
...
```

  If you use SRCH_CALC_ONLY with the PCM_OPFLG_SRCH_CALC_RESULTS flag set, each PIN_FLD_AMOUNT value is returned in its own PIN_FLD_RESULTS array. For example:

```
0 PIN_FLD_RESULTS      ARRAY [0]
1   PIN_FLD_AMOUNT   DECIMAL [0]
0 PIN_FLD_RESULTS      ARRAY [1]
1   PIN_FLD_AMOUNT   DECIMAL [0]
....
```

Using the PIN_FLD_PARAMETERS Field

When you use a predefined search template, you can use an optional **$1** object type parameter in the **from** clause of the search query. This parameter specifies a subclass and allows you to specialize the search without having to modify the stored template.

You use a PIN_FLD_PARAMETERS field when you use the **$1** optional parameter. The **$1** parameter in the search template is replaced by the value of the PIN_FLD_PARAMETERS field on the input flist.

For example, if your template search query is this:

"select X from /device/$1 where F1 = V1 "

and your input flist contains this:

```
...
0 PIN_FLD_ARGS       ARRAY [1] allocated 20, used 1
1   PIN_FLD_POID      POID [0] 0.0.0.1 /device/sim -1 0
0 PIN_FLD_PARAMETERS   STR [0] "sim"
...
```

The string "sim" is substituted for the $1 parameter in the search template and the search looks for the arguments in the **/device/sim** storable class.

> **Note:**
>
> - Using this field does not restrict the search to objects of the type it specifies. To restrict the search, specify the storable class type POID(s) in a PIN_FLD_ARGS array.
> - Be sure to format the value of PIN_FLD_PARAMETERS on the input flist correctly. For example, a value of **portal\user_info** on the input flist does not work, and the search fails without returning an error message. However, when using **portal\user_info** without the trailing '****', the search succeeds.

Limiting Search Results by Using Row Numbers

You use the PIN_FLD_MIN_ROW and PIN_FLD_MAX_ROW fields in the input flist of the PCM_OP_SEARCH opcode to retrieve the records from the search result set using the start row and end row numbers. For example, if the PCM_OP_SEARCH opcode returns 1000 records in the search result set, and PIN_FLD_MIN_ROW is set to 200 and PIN_FLD_MAX_ROW is set to 300, the records from row number 200 to 300 are retrieved.You can navigate through the search result set both in forward and backward directions. For example, after retrieving the records from 200 to 300, you can step backward through the search result set to retrieve the records from 100 to 200.After you run the PCM_OP_SEARCH opcode, any modified records in the database are retrieved from the search result set the next time you run the PCM_OP_SEARCH opcode.

> **Note:**
>
> Opcodes that support multiple database schemas (for example, PCM_OP_GLOBAL_SEARCH and PCM_OP_GLOBAL_STEP_SEARCH) do not support retrieving the search results using row numbers.

Using the "in" Operator

Using the SQL *in* operator in your query is another way of simplifying your search criteria. If you use *in*, you must use the BRM syntax requirements for this operator:

> **Note:**
>
> When using the *in* operator, only POIDs and strings can be searched and POIDs must be type only.

For example, "select X from /config where F1 in (V1 , '/config/locales_map') "

The **where** clause syntax must be entered exactly as shown and follow these requirements:

- There must be one space before *in*.
- There must be one space after *in* and also after the parenthesis following *in*.
- There must be one space between **V1** and the following comma.

> **Note:**
>
> The space between **V1** and the comma is only required when using the *in* operator.

- All values in the query using an *in* operator must be inside parenthesis.

  For example:

  ```
  ...where F1 in ( V1 , '/config/locales_map' ) and F2 in ( V2 ) "
  ```

A search using the following input flist will return the entire contents of **/config/notify** and **/config/locales_map**:

```
0 PIN_FLD_POID     POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_FLAGS     INT [0] 256
0 PIN_FLD_TEMPLATE  STR [0] "select X from /config where F1 in (V1 , '/config/
locales_map' )"
0 PIN_FLD_ARGS    ARRAY [1] allocated 20, used 1
1   PIN_FLD_POID   POID [0] 0.0.0.1 /config/notify -1 0
0 PIN_FLD_RESULTS ARRAY [*] NULL array ptr
```

Searching Subclasses

Because a subclass inherits the attributes of its parent storable class, a simple search includes results from all subclasses of the storable class specified in the arguments array, provided they match the search criteria. You must specify only the most derived class which has a referenced argument in the **where** clause.

To constrain your search to criteria that only exist in a subclass, you must use this subclass on the query itself.

For example, the **/config/notify** object contains a PIN_FLD_EVENTS array, but its parent storable class, **/config**, does not. If your search argument is contained in the PIN_FLD_EVENTS array of the **/config/notify** object, but you specify the **/config** object, the search will fail.

The *faulty* flist looks like this:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /config where F1 = V1 "
0 PIN_FLD_ARGS          ARRAY [1] allocated 20, used 1
1   PIN_FLD_EVENTS      ARRAY [0] allocated 20, used 1
2      PIN_FLD_TYPE_STR   STR [0] "/event/session"
0 PIN_FLD_RESULTS       ARRAY [*] NULL array ptr
```

The *correct* flist specifies the subclass containing the argument:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_FLAGS           INT [0] 256
0 PIN_FLD_TEMPLATE        STR [0] "select X from /config/notify where F1 = V1 "
0 PIN_FLD_ARGS          ARRAY [1] allocated 20, used 1
1   PIN_FLD_EVENTS      ARRAY [0] allocated 20, used 1
2      PIN_FLD_TYPE_STR   STR [0] "/event/session"
0 PIN_FLD_RESULTS       ARRAY [*] NULL array ptr
```

Returning Specific Storable Classes

You must specify the objects you want returned in the PIN_FLD_ARGS array in the search input flist. A search will return objects of the superclass or any other derived class. To return only objects from the specified storable class, the POID of the storable class type must be **-1**.

To return only one specified storable class, include it in the arguments array:

```
0 PIN_FLD_TEMPLATE    STR [0] "select X from /device/sim where F1 = V1 "
0 PIN_FLD_ARGS      ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID     POID [0] 0.0.0.1 /device/sim -1 0
...
```

To return more than one storable class type, but restrict the results to only those types specified, add the storable class type POID for each type to return to the PIN_FLD_ARGS array and set each POID to **-1**:

```
0 PIN_FLD_TEMPLATE    STR [0] "select X from /device/sim where F1 = V1 "
0 PIN_FLD_ARGS      ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID     POID [0] 0.0.0.1 /device/sim -1 0
0 PIN_FLD_ARGS      ARRAY [2] allocated 1, used 1
1   PIN_FLD_POID     POID [0] 0.0.0.1 /device/num -1 0
...
```

To return only a specific storable class and all its subclasses, add the POID of the parent storable class type to the argument list, set the POID to **-1**, use a **like** operator in the **where** clause, and add a percent sign (%) at the end of the storable class type:

> **Note:**
>
> The like operator is only used when searching for strings.

```
0 PIN_FLD_TEMPLATE    STR [0] "select X from /device/sim where F1 like V1 "
0 PIN_FLD_ARGS      ARRAY [1] allocated 1, used 1
1   PIN_FLD_POID     POID [0] 0.0.0.1 /device/sim% -1 0
...
```

Returning Entire Arrays

Because arrays are fields in storable classes, to return an array you must add it to the PIN_FLD_RESULTS array in the search opcode input flist. To return the entire contents of an array, you specify the array and give it a NULL value. Note that a NULL array is different from an empty array in which elements are allocated but not used.

For example, to retrieve the entire contents of the PIN_FLD_NAMEINFO array from the ***/account*** object, use the following input flist:

```
0 PIN_FLD_POID        POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_FLAGS        INT [0] 256
0 PIN_FLD_TEMPLATE     STR [0] "select X from /account where F1 = V1 "
0 PIN_FLD_ARGS       ARRAY [1] allocated 20, used 1
1   PIN_FLD_POID      POID [0] 0.0.0.1 /account 12345 0
0 PIN_FLD_RESULTS    ARRAY [*]     NULL array ptr
1   PIN_FLD_NAMEINFO ARRAY [*] NULL  (instead of: ALLOCATED 20, USED 0 )NULL
```

When constructing this flist in your application, to add a NULL array you must use PIN_FLIST_ELEM_SET instead of PIN_FLIST_ELEM_ADD.

For example:

```
PIN_FLIST_ELEM_SET(flistp, NULL, PIN_FLD_NAMEINFO, PCM_RECID_ALL, ebufp);
```

This entry is *incorrect* as it adds an empty array instead of a NULL array:

```
PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_NAMEINFO, PCM_RECID_ALL, ebufp);
```

Search Template Examples

You must provide all arguments for each search template. The following logic is used when a search is performed:

```
If the results flist is a NULL flist {
    if CM_OPFLG_COUNT_ONLY is set {
        return the count of matched objects (search count)
    } else {
        return each of the entire objects that matches
          (search robj)
    }
} else {
    If this is a calculated search {
        return the result of the calculation
    } else {
        for each object matched, return just those fields
          specified on the RESULTS flist
    }
}
```

Using a Predefined Template

The following example is a predefined search template that you store in the database as a ***/search*** object. The search query placeholders are replaced by the values specified in the arguments array in the input flist, which determine the objects you search for.

```
-- 301 -- 2 arg = search in /pop
insert into search_t (
        poid_db, poid_type, poid_id0, poid_rev,
        name,
        created_t, mod_t,
        --
        flags,
        template
) values (
        DB_NO, '/search', 301, 1,
        '2 arg = search in /pop',
        DATE, DATE,
        --
        SRCH_DISTINCT,
```

```
        'select X from /pop where F1 = V1 and F2 = V2 '
);
```

The template ID is **301** and the **where** clause has 2 arguments. You must enter the search template in the database with the valid date values and database number. Your search template will not work unless it is in the database. When your application performs a search, it calls ID **#301** and your flist must contain the two arguments.

To create a predefined search template:

1. Create a search flist that specifies the template. This example shows how to create a search flist that specifies the above template:

```
/*******************************************************
         * Allocate the flist for searching.
*******************************************************/
flistp = PIN_FLIST_CREATE(ebufp);

/*******************************************************
 * Get the database number.
*******************************************************/
poidp = (poid_t *)PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_POID,0, ebufp);
database = PIN_POID_GET_DB(poidp);

/*******************************************************
* Use 301, the 2 arg search for pop objects.
*******************************************************/
vp = PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_ANI, 1, ebufp);
id = (u_int64)301;
objp = PIN_POID_CREATE(database, "/search", id, ebufp);
PIN_FLIST_FLD_PUT(flistp, PIN_FLD_POID, (void *)objp, ebufp);

/*******************************************************
* Return pop that matches ani and is a primary pop.
*******************************************************/
a_flistp = PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_ARGS, 1, ebufp);
aniarray_flistp = PIN_FLIST_CREATE(ebufp);
/*
** PIN_FLD_ANI for our first arg.
*/
PIN_FLIST_FLD_SET(aniarray_flistp, PIN_FLD_ANI, vp, ebufp);
PIN_FLIST_ELEM_SET(a_flistp, aniarray_flistp, PIN_FLD_ANIS, 0,ebufp );
/*
** PIN_FLD_TYPE for our second arg.
*/
a_flistp = PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_ARGS, 2, ebufp);
aniarray_flistp = PIN_FLIST_CREATE(ebufp);
PIN_FLIST_FLD_SET(aniarray_flistp, PIN_FLD_TYPE, (void *)&type,ebufp);
PIN_FLIST_ELEM_SET(a_flistp, aniarray_flistp,PIN_FLD_ANIS, 0, ebufp);
PIN_DESTROY_FLIST(aniarray_flistp,ebufp);

/*******************************************************
* Put on the PIN_FLD_RESULTS array for our results.
*******************************************************/
PIN_FLIST_ELEM_SET(flistp, (void *)NULL, PIN_FLD_RESULTS, -1, ebufp);

/*******************************************************
* Call the DM to do the search.
*******************************************************/
PCM_OP(ctxp, PCM_OP_SEARCH, 0, flistp, &r_flistp, ebufp);
```

The search flist fields passed in might look like this:

```
0 PIN_FLD_POID      POID [0] 0.0.0.2 /search 301 0
0 PIN_FLD_ARGS     ARRAY [1] allocated 20, used 1
1   PIN_FLD_ANIS   ARRAY [0] allocated 20, used 1
2      PIN_FLD_ANI   STR [0] "408343"
0 PIN_FLD_ARGS     ARRAY [2] allocated 20, used 1
1   PIN_FLD_ANIS   ARRAY [0] allocated 20, used 1
2      PIN_FLD_TYPE  INT [0] 4
0 PIN_FLD_RESULTS  ARRAY [*] Null pointer
```

2. Load the search template into the database. You can use **testnap** to load the search template into the database. For more information, see "Creating a New Search Object".
3. Add a copy of the new search template to the **init_objects.source** file.

When the Storage Manager receives the flist, it first queries the database to find the template with **poid_id 301**. When it has the template, it looks for the PIN_FLD_ARGS array on your input flist and substitutes field numbers with field names and values with the values passed in. It then performs the search and returns the matching objects from the database.

In this example, no specific fields were specified to be returned, so the entire object that matches the search criteria is returned.

Defining the Search Template at Runtime

You can perform a search without using a predefined, stored template by including a template field in the search flist.

This example shows how to create a search flist that specifies a runtime search template:

```
/*******************************************************
         * Allocate the flist for searching.
*******************************************************/
flistp = PIN_FLIST_CREATE(ebufp);
char * template = "select X from /pop where F1 = V1 and F2 = V2 "

/*******************************************************
 * Get the database number.
*******************************************************/
poidp = (poid_t *)PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_POID,0, ebufp);
database = PIN_POID_GET_DB(poidp);

/*******************************************************
* Use -1, the 2 arg search for pop objects.
*******************************************************/
vp = PIN_FLIST_FLD_GET(in_flistp, PIN_FLD_ANI, 1, ebufp);
id = -1;
objp = PIN_POID_CREATE(database, "/search", id, ebufp);
PIN_FLIST_FLD_PUT(flistp, PIN_FLD_POID, (void *)objp, ebufp);
PIN_FLIST_FLD_SET (in_flistp, PIN_FLD_TEMPLATE, template, ebufp)

/*******************************************************
* Return pop that matches ani and is a primary pop.
*******************************************************/
a_flistp = PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_ARGS, 1, ebufp);
aniarray_flistp = PIN_FLIST_CREATE(ebufp);
/*
** PIN_FLD_ANI for our first arg.
*/
PIN_FLIST_FLD_SET(aniarray_flistp, PIN_FLD_ANI, vp, ebufp);
PIN_FLIST_ELEM_SET(a_flistp, aniarray_flistp, PIN_FLD_ANIS, 0,ebufp );
/*
```

```
** PIN_FLD_TYPE for our second arg.
*/
a_flistp = PIN_FLIST_ELEM_ADD(flistp, PIN_FLD_ARGS, 2, ebufp);
aniarray_flistp = PIN_FLIST_CREATE(ebufp);
PIN_FLIST_FLD_SET(aniarray_flistp, PIN_FLD_TYPE, (void *)&type,ebufp);
PIN_FLIST_ELEM_SET(a_flistp, aniarray_flistp,PIN_FLD_ANIS, 0, ebufp);
PIN_DESTROY_FLIST(aniarray_flistp,ebufp);

/*********************************************************
* Put on the PIN_FLD_RESULTS array for our results.
*********************************************************/
PIN_FLIST_ELEM_SET(flistp, (void *)NULL, PIN_FLD_RESULTS, -1, ebufp);

/*********************************************************
* Call the DM to do the search.
*********************************************************/
PCM_OP(ctxp, PCM_OP_SEARCH, 0, flistp, &r_flistp, ebufp);
```

The search flist fields passed in might look like this:

```
0 PIN_FLD_POID       POID [0] 0.0.0.2 /search/pop -1 0
0 PIN_FLD_ARGS      ARRAY [1] allocated 20, used 1
1  PIN_FLD_ANIS      ARRAY[0] allocated 20, used 1
2      PIN_FLD_ANI    STR [0] "408343"
0 PIN_FLD_ARGS      ARRAY [2] allocated 20, used 1
1   PIN_FLD_ANIS     ARRAY[0] allocated 20, used 1
2      PIN_FLD_TYPE   INT [0] 4
0 PIN_FLD_RESULTS   ARRAY [*] Null pointer
0 PIN_FLD_TEMPLATE    STR [0] select X from /pop where F1 = V1 and F2 = V2
```

With the template field in the flist, the Storage Manager looks for the PIN_FLD_ARGS array on your input flist and substitutes field numbers with field names and values with the values passed in. It then performs the search and returns the matching objects from the database. Note that the arguments in the PIN_FLD_ARGS array must be the specified storable classes and not the superclass.

In this example, no specific fields were specified to be returned, so the entire object that matches the search criteria is returned.

About Single-Schema Searches

There are two basic ways to search for information in a single database schema:

- Search the schema and return all of the results at once by calling PCM_OP_SEARCH.
- Search the schema with PCM_OP_STEP_SEARCH, which uses PCM_OP_STEP_NEXT and PCM_OP_STEP_END to display the results as smaller sets of accounts.

The searches performed by PCM_OP_SEARCH and PCM_OP_STEP_SEARCH are identical; they take the same input flist and return the same results. The results of PCM_OP_SEARCH, however, can be very large—large enough to use all the DM shared memory. If you expect the size of your search results to be very large, use the PCM_OP_STEP_SEARCH opcode for the search. Step searching has the following advantages:

- **Speed**: The search results come back much faster in smaller pieces. PCM_OP_STEP_SEARCH returns the first set of results immediately; it does not wait for the entire result set to be built.

- **System Resources**: PCM_OP_STEP_SEARCH allocates just enough shared memory in the DM for a single set of results at a time. PCM_OP_SEARCH results use enough shared memory for the entire result set all at once.

The disadvantage to step searching is that you must call all three step search opcodes for each search: PCM_OP_STEP_SEARCH, PCM_OP_STEP_NEXT, and PCM_OP_STEP_END.

Performing a Search on a Single Schema

To perform a search on a single database schema, use the PCM_OP_SEARCH opcode.

This opcode enables a client application to search for objects that meet a set of criteria defined by the client application.

If two objects have an encrypted field that contains the same data encrypted with different keys, a PCM_OP_SEARCH for that value returns only one object.

> **Note:**
>
> Use this opcode only to search a single, known schema. If your BRM implementation uses multiple schemas and you must search more than one, use the PCM_OP_GLOBAL_SEARCH opcode.

When using the PCM_OP_SEARCH opcode, you can apply the *order by* clause only to the top-level arrays. The *order by* clause cannot be applied to subarrays.

For information about required fields in the input flist, see "About the Search Input Flist".

This opcode performs a search by creating a search template at runtime. To use a stored template instead, the POID must specify a template **/search** object. If it is specified, this opcode searches for a stored template in the database schema and uses that template for the search.

> **Note:**
>
> Performing a search using a **/search** object template stored in the database is supported but not recommended.

Search results can be manipulated by using the flist field handling macros. For a list of opcodes, see Flist Field-Handling Macros in *BRM Developer's Reference*.

Flags

These flags are used in the call to the PCM_OP_SEARCH opcode:

- To return only the number of matching results, use PCM_OPFLG_COUNT_ONLY. See "Flags".
- To increase performance when reading fields and objects, use PCM_OPFLG_CACHEABLE. See "Improving Performance when Working with Objects".

> **Note:**
>
> When this flag is set, PCM_OP_SEARCH caches data, which improves performance when that data is read by PCM_OP_READ_FLDS or PCM_OP_READ_OBJ. However, PCM_OP_SEARCH always reads from the database, not from the cache.

Memory Management

If your search returns a large amount of data, you must make sure sufficient memory is available to hold that data. To control the size of the data returned, use the PCM_OP_STEP_SEARCH opcode.

For a discussion of when to use searching and step searching, see "About Single-Schema Searches".

For a discussion of the memory implications of searching, see "The Impact of Searches on Shared Memory Allocation".

Examples

For an example of a simple search, see "Simple Search Example".

For examples of input flists for complex searches, see "Complex Searches".

For a sample search program that searches for a single and multiple results, see *BRM_home*/**apps/sample/sample_search.c**.

Performing a Step Search on a Single Schema

To perform a step search on a single database schema, use the PCM_OP_STEP_SEARCH opcode.

> **Note:**
>
> Use this opcode only to search a single, known schema. If your BRM implementation uses multiple schemas and you must search more than one, use the opcode PCM_OP_GLOBAL_SEARCH.

This opcode enables a client application to define search criteria, search for objects using those criteria, and receive a specified number of result sets. The advantage of using this opcode instead of PCM_OP_SEARCH is that the results are returned in discrete chunks, which enables you to control resource usage in both the DM and the application.

For information on when to use PCM_OP_SEARCH and PCM_OP_STEP_SEARCH, see "About Single-Schema Searches".

This opcode must be used in combination with the PCM_OP_STEP_NEXT and PCM_OP_STEP_END opcodes to complete a search cycle. The cycle must start with PCM_OP_STEP_SEARCH, which initiates a step search and gets the first set of PIN_FLD_RESULT elements. One or more PCM_OP_STEP_NEXT opcodes follow, each retrieving the next specified number of result sets. PCM_OP_STEP_END must come last to end the step search.

> **Note:**
>
> Stepping backward through the result set is not supported.

When a step search is initiated, no other functions can be performed, including another step search, until the search cycle is completed. If a second PCM_OP_STEP_SEARCH opcode is sent to the database before a first has finished its search cycle, an error is returned to the client.

The search criteria are passed in by the client application on the input flist. The input flist must contain a POID, and its type must be **/search**. The POID is ignored. You must also include a PIN_FLD_RESULTS array that indicates which fields and how many matching results to return for this opcode.

In the PIN_FLD_RESULTS_LIMIT field, specify the maximum number of results to be returned from all steps of the search. The information from this field is conveyed to the database so that the search is run more efficiently. If this field is not specified, all matching results are cached, even if they are not returned.

> **Note:**
>
> If the search uses an **order by** clause, the PIN_FLD_RESULTS_LIMIT field causes incorrect sorting. Do not use the PIN_FLD_RESULTS_LIMIT field if the search includes an **order by** clause.

Specify search arguments in the PIN_FLD_ARGS array on the input flist. Each element of the array contains one argument for the search. You must provide at least one argument. The maximum number of search arguments is 32. Indicate which argument is contained in an array element sub-flist by specifying the element ID. For example, element-ID 1 corresponds to argument 1.

Search results can be manipulated by using the flist field handling macros. For a list of opcodes, see "Flist Field-Handling Macros" in *BRM Developer's Reference*.

To increase performance, use PCM_OPFLG_CACHEABLE. See "Improving Performance when Working with Objects".

Examples

For an example of input and return flists for step searching, see "Step Search Example".

For examples of input flists for complex searches, see "Complex Searches".

For a sample search program that searches for a single and multiple results, see *BRM_home*/**apps/sample/sample_search.c**.

Getting the Next Set of Search Results from a Step Search

This opcode enables a client application to receive the next set of results from a search initiated by PCM_OP_STEP_SEARCH. Results of the search are returned in discrete chunks.

This opcode must be used in combination with the PCM_OP_STEP_SEARCH and PCM_OP_STEP_END opcodes to complete the step search cycle. PCM_OP_STEP_SEARCH

initiates step searching and gets the first set of PIN_FLD_RESULT elements. PCM_OP_STEP_NEXT goes to the Data Manager and gets the next set of PIN_FLD_RESULT elements. PCM_OP_STEP_END ends the step search.

Use this opcode for each set of results to be returned. Specify the maximum number of records to return as the element-ID of the PIN_FLD_RESULTS element. To return no records, use zero. PCM_OP_STEP_END can be called at any time to end the search.

This opcode uses the same input flist as PCM_OP_STEP_SEARCH.

To increase performance, use the PCM_OPFLG_CACHEABLE flag. See "Improving Performance when Working with Objects".

Ending a Step Search

To end a search result that has been initiated by PCM_OP_STEP_SEARCH, use the PCM_OP_STEP_END opcode.

This opcode must be used in combination with the PCM_OP_STEP_SEARCH and PCM_OP_STEP_NEXT opcodes to complete the step search cycle. PCM_OP_STEP_SEARCH initiates step searching and gets the first set of PIN_FLD_RESULT elements. PCM_OP_STEP_NEXT retrieves the next specified number of results. PCM_OP_STEP_END ends the step search.

Simple Search Example

This example of a simple PCM_OP_SEARCH searches for each account whose status is active and retrieves all corresponding events created in the last week:

```
elem_id = 0;
cookie = (pin_cookie_t)NULL;
while ((acct_flistp = PIN_FLIST_ELEM_GET_NEXT(flistp,
                        PIN_FLD_RESULTS, &elem_id, 1, &cookie, ebufp))
                != (pin_flist_t *)NULL) {

/* get the status of the current account */
status = PIN_FLIST_FLD_GET(acct_flist, PIN_FLD_STATUS, 0, ebufp);

/* process accordingly, based on status */
switch (status) {
case PIN_STATUS_ACTIVE:
/* fetch events created in the last week */
fetch_last_weeks_events(cur_flist, ebufp);
break;
case PIN_STATUS_INACTIVE:
/* do something */
break;
default:
/* log an error */
break;
}
}

fetch_last_weeks_events(acct_flist, ebufp)
{
  /*
         * Create the search flist.
         */
        s_flistp = PIN_FLIST_CREATE(ebufp);
```

```
        /*
         * Create and add the search poid.
         */
        search_poidp = PIN_POID_CREATE((int64)0, "/search", (int64)-1, ebufp);
        PIN_FLIST_FLD_PUT(s_flistp, PIN_FLD_POID, (void *)search_poidp, ebufp);

        /*
         * Add the search template.
         */
        PIN_FLIST_FLD_PUT(s_flistp, PIN_FLD_TEMPLATE,
                (void *)"select X from /event where F1 = V1 and F2 > V2 ", ebufp);

        /*
         * Add the search arguments.
         */
        arg_flistp = PIN_FLIST_ELEM_ADD(s_flistp, PIN_FLD_ARGS, 1, ebufp);
        acct_poidp = PIN_FLIST_FLD_TAKE(acct_flist, PIN_FLD_POID, 0, ebufp);
        PIN_FLIST_FLD_PUT(arg_flistp, PIN_FLD_ACCOUNT_OBJ, acct_poidp, ebufp);

        arg_flistp = PIN_FLIST_ELEM_ADD(s_flistp, PIN_FLD_ARGS, 2, ebufp);
  one_week_ago = <timestamp corresponding to 1 week ago>;
        PIN_FLIST_FLD_PUT(arg_flistp, PIN_FLD_CREATED_T, &one_week_ago, ebufp);


        /*
         * Fetch everything.
         */
        PIN_FLIST_FLD_PUT(s_flistp, PIN_FLD_RESULT, (void *)NULL, ebufp);

        /*
         * Do the search.
         */
        PCM_OP(pcm_ctxp, PCM_OP_SEARCH, PCM_OPFLG_READ_UNCOMMITTED,
                s_flistp, &r_flistp, ebufp);

/*
 * do something with the events we just fetched
 */
}
```

Step Search Example

This example shows the results of a call to PCM_OP_STEP_SEARCH. This step search has four steps.

> **Note:**
>
> At the end of a step search, you do *not* receive a results array. You receive only your search POID, as shown at the end of this example.

Input flist:

```
0 PIN_FLD_POID               POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_TEMPLATE            STR [0] "select X from /account where F1 like V1 "
0 PIN_FLD_FLAGS               INT [0] 0
0 PIN_FLD_ARGS              ARRAY [1]
1   PIN_FLD_NAMEINFO        ARRAY [*]
2     PIN_FLD_FIRST_CANON     STR [0] "%"
```

```
0 PIN_FLD_RESULTS           ARRAY [4]
1   PIN_FLD_ACCOUNT_NO        STR [0] ""
```

Search results:

```
# number of field entries allocated 5, used 5
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS       ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO    STR [0] "ROOT.0.0.1"
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account 1 1
0 PIN_FLD_RESULTS       ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO    STR [0] "0.0.0.1-8759"
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account 8759 133
0 PIN_FLD_RESULTS       ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO    STR [0] "0.0.0.1-9267"
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account 9267 122
0 PIN_FLD_RESULTS       ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO    STR [0] "0.0.0.1-9961"
1   PIN_FLD_POID         POID [0] 0.0.0.1 /account 9961 128
```

Results of the calls to PCM_OP_STEP_NEXT:

```
# number of field entries allocated 5, used 5
0 PIN_FLD_POID            POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-10709"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 10709 53
0 PIN_FLD_RESULTS        ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-10721"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 10721 98
0 PIN_FLD_RESULTS        ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-10881"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 10881 134
0 PIN_FLD_RESULTS        ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-11057"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 11057 122

# number of field entries allocated 5, used 5
0 PIN_FLD_POID            POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12047"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12047 75
0 PIN_FLD_RESULTS        ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12213"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12213 123
0 PIN_FLD_RESULTS        ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12241"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12241 122
0 PIN_FLD_RESULTS        ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12356"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12356 39

# number of field entries allocated 5, used 5
0 PIN_FLD_POID            POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12484"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12484 45
0 PIN_FLD_RESULTS        ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12569"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12569 8
0 PIN_FLD_RESULTS        ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12590"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12590 19
```

```
0 PIN_FLD_RESULTS        ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12612"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12612 12

# number of field entries allocated 5, used 5
0 PIN_FLD_POID            POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12697"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12697 8
0 PIN_FLD_RESULTS        ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12705"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12705 14
0 PIN_FLD_RESULTS        ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-12740"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 12740 54
0 PIN_FLD_RESULTS        ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-13090"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 13090 38

# number of field entries allocated 5, used 5
0 PIN_FLD_POID            POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-13346"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 13346 45
0 PIN_FLD_RESULTS        ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-13476"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 13476 48
0 PIN_FLD_RESULTS        ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-13732"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 13732 66
0 PIN_FLD_RESULTS        ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO     STR [0] "0.0.0.1-13956"
1   PIN_FLD_POID          POID [0] 0.0.0.1 /account 13956 84

# number of field entries allocated 5, used 5
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS          ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO       STR [0] "0.0.0.1-14313"
1   PIN_FLD_POID            POID [0] 0.0.0.1 /account 14313 8
0 PIN_FLD_RESULTS          ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO       STR [0] "0.0.0.1-14825"
1   PIN_FLD_POID            POID [0] 0.0.0.1 /account 14825 8
0 PIN_FLD_RESULTS          ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO       STR [0] "0.0.0.1-14896"
1   PIN_FLD_POID            POID [0] 0.0.0.1 /account 14896 70
0 PIN_FLD_RESULTS          ARRAY [3] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO       STR [0] "0.0.0.1-15069"
1   PIN_FLD_POID            POID [0] 0.0.0.1 /account 15069 12

# number of field entries allocated 5, used 5
0 PIN_FLD_POID             POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_RESULTS         ARRAY [0] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO      STR [0] "0.0.0.1-15129"
1   PIN_FLD_POID           POID [0] 0.0.0.1 /account 15129 8
0 PIN_FLD_RESULTS         ARRAY [1] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO      STR [0] "0.0.0.1-15257"
1   PIN_FLD_POID           POID [0] 0.0.0.1 /account 15257 8
0 PIN_FLD_RESULTS         ARRAY [2] allocated 2, used 2
1   PIN_FLD_ACCOUNT_NO      STR [0] "0.0.0.1-15385"
1   PIN_FLD_POID           POID [0] 0.0.0.1 /account 15385 8
0 PIN_FLD_RESULTS         ARRAY [3] allocated 2, used 2
```

```
1   PIN_FLD_ACCOUNT_NO      STR [0] "0.0.0.1-15824"
1   PIN_FLD_POID           POID [0] 0.0.0.1 /account 15824 68
```

Final set of search results:

```
# number of field entries allocated 1, used 1
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search 0 0
```

> **Note:**
>
> You do not receive a results array with the last call to PCM_OP_STEP_NEXT because there are no more search results. You call PCM_OP_STEP_END to finish the search.

Results of the call to PCM_OP_STEP_END:

```
# number of field entries allocated 3, used 3
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_ARGS             ARRAY [1] allocated 1, used 1
1   PIN_FLD_NAMEINFO       ARRAY [0] allocated 1, used 1
2      PIN_FLD_FIRST_CANON   STR [0] "%"
0 PIN_FLD_RESULTS          ARRAY [4] allocated 1, used 1
1   PIN_FLD_ACCOUNT_NO       STR [0] ""
```

Performing Exact Searches

To search and return array elements, you use an exact search. Exact searches enable you to limit results to only the array elements that match the search criteria.

If your search includes a **where** clause and you want that clause to be applied to array elements, use the SRCH_EXACT (**512**) flag with the PCM_OP_SEARCH and PCM_OP_STEP_SEARCH opcodes. If you do not use the SRCH_EXACT flag, the **where** clause is applied to the object and not limited to array elements that match the search criteria within that object. The results, therefore, include all items in the object instead of only those items in the array that match your search criteria.

For example, suppose you want to search for all **/rate** objects with general ledger IDs greater than 0 and element IDs less than 1001. *Without* the SRCH_EXACT flag, the input flist looks like this:

```
0 PIN_FLD_POID                    POID [0] 0.0.0.1 /search 0 0
0 PIN_FLD_TEMPLATE                 STR [0] "select X from /rate where F1 > V1 and F2 <
V2 "
0 PIN_FLD_FLAGS                    INT [0] 256

0 PIN_FLD_ARGS                   ARRAY [1] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 20, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 20, used 1
3         PIN_FLD_GL_ID            INT [0] 0

0 PIN_FLD_ARGS                   ARRAY [2] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 20, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 20, used 1
3         PIN_FLD_ELEMENT_ID       INT [0] 1001

0 PIN_FLD_RESULTS                ARRAY [0] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 20, used 1
```

```
2      PIN_FLD_BAL_IMPACTS        ARRAY [0] allocated 20, used 2
3         PIN_FLD_ELEMENT_ID        INT [0] 0
3         PIN_FLD_GL_ID             INT [0] 0
```

The output flist might look like this:

```
0 PIN_FLD_POID                    POID [0] 0.0.0.1 /search 0 0

0 PIN_FLD_RESULTS                ARRAY [0]allocated 2, used 2
1   PIN_FLD_POID                  POID [0] 0.0.0.1 /rate 8257 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID       INT [0] 978
3         PIN_FLD_GL_ID            INT [0] 13000001

0 PIN_FLD_RESULTS                ARRAY [1] allocated 2, used 2
1   PIN_FLD_POID                  POID [0] 0.0.0.1 /rate 8321 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID       INT [0] 978
3         PIN_FLD_GL_ID            INT [0] 12000065

0 PIN_FLD_RESULTS                ARRAY [3] allocated 2, used 2
1   PIN_FLD_POID                  POID [0] 0.0.0.1 /rate 8702 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID       INT [0] 978
3         PIN_FLD_GL_ID            INT [0] 1000001

0 PIN_FLD_RESULTS                ARRAY [2] allocated 2, used 2
1   PIN_FLD_POID                  POID [0] 0.0.0.1 /rate8446 1
1   PIN_FLD_QUANTITY_TIERS       ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS       ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID       INT [0] 1000006
3         PIN_FLD_GL_ID            INT [0] 51000001
```

> **Note:**
>
> The last result is incorrect; the element ID is greater than 1001. Incorrect results such as this occur because the search finds all the objects that match the condition in the **where** clause but does not apply that clause to the array elements.

If you use SRCH_EXACT by entering a flag value of **512**, the input flist looks like this:

```
0 PIN_FLD_POID                     POID [0] 0.0.0.1 /search 0 0

0 PIN_FLD_TEMPLATE                  STR [0] "select X from /rate where F1 > V1 and F2 <
V2 "

0 PIN_FLD_FLAGS                     INT [0] 512

0 PIN_FLD_ARGS                     ARRAY [1] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS         ARRAY [0] allocated 20, used 1
2      PIN_FLD_BAL_IMPACTS         ARRAY [0] allocated 20, used 1
3         PIN_FLD_GL_ID              INT [0] 0

0 PIN_FLD_ARGS                     ARRAY [2] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS         ARRAY [0] allocated 20, used 1
2      PIN_FLD_BAL_IMPACTS         ARRAY [0] allocated 20, used 1
```

```
3         PIN_FLD_ELEMENT_ID         INT [0] 1001

0 PIN_FLD_RESULTS                  ARRAY [0] allocated 20, used 1
1   PIN_FLD_QUANTITY_TIERS         ARRAY [0] allocated 20, used 1
2      PIN_FLD_BAL_IMPACTS         ARRAY [0] allocated 20, used 2
3         PIN_FLD_ELEMENT_ID         INT [0] 0
3         PIN_FLD_GL_ID              INT [0] 0
```

The output flist might look like this:

```
0 PIN_FLD_POID                     POID [0] 0.0.0.1 /search 0 0

0 PIN_FLD_RESULTS                 ARRAY [0]allocated 2, used 2
1   PIN_FLD_POID                   POID [0] 0.0.0.1 /rate 8257 1
1   PIN_FLD_QUANTITY_TIERS        ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS        ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID        INT [0] 978
3         PIN_FLD_GL_ID             INT [0] 13000001

0 PIN_FLD_RESULTS                 ARRAY [1] allocated 2, used 2
1   PIN_FLD_POID                   POID [0] 0.0.0.1 /rate 8321 1
1   PIN_FLD_QUANTITY_TIERS        ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS        ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID        INT [0] 978
3         PIN_FLD_GL_ID             INT [0] 12000065

0 PIN_FLD_RESULTS                 ARRAY [2] allocated 2, used 2
1   PIN_FLD_POID                   POID [0] 0.0.0.1 /rate 8702 1
1   PIN_FLD_QUANTITY_TIERS        ARRAY [0] allocated 1, used 1
2      PIN_FLD_BAL_IMPACTS        ARRAY [0] allocated 2, used 2
3         PIN_FLD_ELEMENT_ID        INT [0] 978
3         PIN_FLD_GL_ID             INT [0] 1000001
```

> **Note:**
>
> This search returns only objects that satisfy both parts of the **where** clause. Spurious results are eliminated.

Using "like" with Exact Searches

You use a **like** operator with an exact search to return all elements of an array that match the search criteria.

The following example returns all arrays from **/account** that contain any string in the PIN_FLD_FIRST_CANNON field:

```
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_TEMPLATE           STR [0] "select X from /account where F1 like V1 "
0 PIN_FLD_ARGS             ARRAY [1]
1   PIN_FLD_NAMEINFO       ARRAY [*]
2      PIN_FLD_FIRST_CANON   STR [0] "%"
0 PIN_FLD_RESULTS          ARRAY [*] NULL array ptr
```

Exact Search Limitations

When performing a complex search (searching across multiple objects), using an array element as a join column is not supported with exact searches. In the **where** clause, trying to

match a value in one table with a value in another table can return results that do not match your search criteria.

For example, to find all charge offers in an account with a charge offer name that begins with the string **"Pr"**, you might use the following query:

```
"select X from /account 1, /product 2 where 1.F1 = V1 and 1.F2 = 2.F3 and 2.F4 like V4 "
```

Where the arguments array is:

```
0 PIN_FLD_ARGS               ARRAY [1] allocated 20, used 1
1   PIN_FLD_POID              POID [0] 0.0.0.1 /account 12345
0 PIN_FLD_ARGS               ARRAY [2] allocated 20, used 1
1   PIN_FLD_PRODUCTS         ARRAY [0]
2      PIN_FLD_PRODUCT_OBJ    POID [0] NULL
0 PIN_FLD_ARGS               ARRAY [3] allocated 20, used 1
1   PIN_FLD_POID              POID [0] NULL
0 PIN_FLD_ARGS               ARRAY [4] allocated 20, used 1
1   PIN_FLD_NAME                 STR [0] "Pr%"
```

However, with an exact search, **"1.F2 = 2.F3"** in the **where** clause causes the search to return *all* charge offers in the account if at least one charge offer name starts with the string **"Pr"**.

An alternative to using an array element as a join column with exact searches is to perform two separate searches and then compare the search results for matching data. In the preceding example, you search **/product** for charge offers that begin with **"Pr"** and then perform the preceding search as shown to return all charge offers for the **/account**. Then, for each POID in the **/product** list, iterate through the **/account** charge offers list to find the matching charge offers.

Complex Searches

You can perform a complex search across multiple objects by including each object in the *object_name* section of the PIN_FLD_TEMPLATE string.

In a simple search, when a client wants a list of all events pertaining to bundle purchases, the client must perform two searches:

- Get all **/deal** object POIDs (with the bundle names).
- Search the event storable class (and subclasses) with each of the **/deal** object POIDs obtained from the previous search for the required event fields.

A complex search eliminates the need for cascading searches and enables the client to issue one complex search instead of multiple simple searches.

> **Note:**
>
> The search results include only the fields from the first storable class in the query.

Use the following rules when creating a complex search template:

- When connecting separate objects types (**1.F3** = **2.F4**), explicitly specify the join clause as part of the template by using the form **X.F***m* **= Y.F***n*.
- **F***m* and **F***n* must have a value specified as a NULL pointer in the PIN_FLD_ARGS array.
- When you use an **order by** clause (which can be included when you have a **where** clause), set the SRCH_DISTINCT flag to 0.

> **Tip:**
>
> If you want to regularly perform distinct searches while using an **order by** clause, you can create a storable class and database view specifically for searching. See "About Performing Distinct Searches with Ordering and Pagination".

- A maximum of six separate objects is allowed in a complex search template.

The following example shows the template syntax to retrieve all bundle purchase events:

"select *X* from **/event/billing/deal 1**, **/deal 2** where (**2.F1** Like **V1** and **2.F2** != **V2** and **1.F3** = **2.F4**) order by **3.F5**"

> **Note:**
>
> There are no stored templates available for complex searches.

Complex Search Example

The following example searches for all bundle purchase events.

Input flist:

```
# number of field entries allocated 5, used 5
0 PIN_FLD_RESULTS           ARRAY [10]
1   PIN_FLD_SYS_DESCR         STR [0] NULL str ptr
1   PIN_FLD_ACCOUNT_OBJ      POID [0] NULL poid pointer
1   PIN_FLD_POID             POID [0] NULL poid pointer
1   PIN_FLD_END_T          TSTAMP [0] (0) <null>
0 PIN_FLD_ARGS              ARRAY [1]
1   PIN_FLD_NAME              STR [0] "%"
0 PIN_FLD_ARGS              ARRAY [2]
1   PIN_FLD_POID             POID [0] 0.0.0.1 /deal 0 0
0 PIN_FLD_ARGS              ARRAY [3]
1   PIN_FLD_DEAL_INFO   SUBSTRUCT [0]
2      PIN_FLD_DEAL_OBJ      POID [0] NULL poid pointer
0 PIN_FLD_ARGS              ARRAY [4]
1   PIN_FLD_POID             POID [0] NULL poid pointer
0 PIN_FLD_TEMPLATE            STR [0] "select X from /event/billing/deal 1, /deal 2
where ( 2.F1 Like V1 and 2.F2 != V2 and 1.F3 = 2.F4 ) "
0 PIN_FLD_FLAGS               INT [0] 256
0 PIN_FLD_POID               POID [0] 0.0.0.1 /search -1 0
```

About Performing Distinct Searches with Ordering and Pagination

Because of the way row numbers are generated for complex searches, you cannot use ordering or pagination when PIN_FLD_FLAGS is set to perform a distinct search (256 or 768).

If you need to make this kind of search frequently, you can create storable classes with a residency type of 9 specifically for searching. Because storable classes of this type are added to the data dictionary without creating any database tables, you can add nested fields from multiple storable classes into a single class with only the required data, without adding more tables to the database. You can then create views for these classes, which have a flat structure

rather than multiple levels of nesting, and perform simple searches on them, rather than relying on complex searches and joins across multiple tables in BRM.

See the following topics:

- Creating Storable Classes and Database Views for Distinct Searches
- Performing Distinct Searches with Ordering and Pagination
- Modifying Storable Classes for Distinct Searches

Creating Storable Classes and Database Views for Distinct Searches

To create the custom storable classes with residency type 9 and database views to perform distinct searches with ordering and pagination:

1. Create a custom storable class using Portal Object Definition Language (PODL) files:
   a. Run the following command for each class that contains the fields you want to use in your new class:

      ```
      pin_deploy class [-smncp] /existing_class
      ```

      where */existing_class* is the existing class, such as **/account** or **/service**.

      This exports the PODL file for the specified class.
   b. Combine the exported PODL files into one, keeping any fields you want in the new class, and giving the file a unique name.
   c. Add the RESIDENCY_TYPE attribute, set to 9 (GLOBAL_DB_VIEW).
   d. Run the following command to import the new PODL file into the BRM database:

      ```
      pin_deploy create new_class.podl
      ```

      where *new_class* is the name of your new class.

      See "Deploying Custom Fields and Storable Class Definitions" and "pin_deploy" for more information about using **pin_deploy**.
2. Create a database view that includes the new storable class and any relevant fields from the base class tables.
   See "CREATE VIEW" in *Oracle Database SQL Language Reference* for more information.

Performing Distinct Searches with Ordering and Pagination

When performing searches in BRM with ordering and pagination on storable classes of residency type 9 and the views created for them:

- Use your new storable class for the *object_name* search element.
- Use your new view in the **where** clause.
- Use the ROWNUM pseudo column to assign row numbers to the search results and use them to paginate the results. See "ROWNUM Pseudocolumn" in *Oracle Database SQL Language Reference*.
- Use the **order by** clause with columns from the new view.

For example, to show results 1-10 in descending order of time created, you would use the following in the PIN_FLD_TEMPLATE field of the search input flist:

```
select X from /new_storable_class where column in ( select column from
( select rownum rid,
column from new_view_t where F1 like V1 and  F2 != V2 )a where a.rid >= 1 and
a.rid<= 10 )
order by new_view_t.created_t desc"
```

where:

- */new_storable_class* is the new storable class you created with residency type 9.
- *column* is the column you want to select, usually containing a POID.
- *new_view_t* is the new database view you created for the new storable class.

Modifying Storable Classes for Distinct Searches

To modify the custom storable classes with residency type 9:

1. Run the following command to export the PODL file for your custom storable class:

   ```
   pin_deploy class [-smncp] /custom_class
   ```

   where */custom_class* is the name of your custom class.
2. Add, remove, or modify the fields in the class.
3. Run the following command to import the modified PODL file and replace the existing version in the BRM database:

   ```
   pin_deploy replace custom_class.podl
   ```

   where *custom_class* is the name of your new class.
4. Recreate the database view for the class. See "CREATE VIEW" in *Oracle Database SQL Language Reference* for more information about these statements.

Search without POID

Use the SEARCH_WITHOUT_POID (**1024**) flag to return data without the POID for each result. To do this, set the value to **1024**.

Without the SRCH_WITHOUT_POID flag set, the input flist looks like this:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_TEMPLATE       STR [0] "select X from /event where F1 like V1 "
0 PIN_FL0 PIN_FLD_ARGS   ARRAY [1] allocated 20, used 1D_FLAGS INT [0] 0
1 PIN_FLD_POID          POID [0] 0.0.0.1 /event/session -1 0
0 PIN_FLD_RESULTS       ARRAY [4] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
```

The output flist might look like this:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_RESULTS        ARRAY [0] allocated 20, used 2
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
```

```
1 PIN_FLD_POID           POID [0] 0.0.0.1 /event/session 252887674388488909 1
0 PIN_FLD_RESULTS        ARRAY [1] allocated 20, used 2
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
1 PIN_FLD_POID           POID [0] 0.0.0.1 /event/session 252887674388490957 1
0 PIN_FLD_RESULTS        ARRAY [2] allocated 20, used 2
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
1 PIN_FLD_POID           POID [0] 0.0.0.1 /event/session 252887674388489103 1
0 PIN_FLD_RESULTS        ARRAY [3] allocated 20, used 2
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
1 PIN_FLD_POID            POID [0] 0.0.0.1 /event/session 252887674388491981 1
```

With the SRCH_WITHOUT_POID flag set, the input flist looks like this:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_TEMPLATE       STR [0] "select X from /event where F1 like V1 "
0 PIN_FLD_FLAGS          INT [0] 1024
0 PIN_FLD_ARGS           ARRAY [1] allocated 20, used 1
1 PIN_FLD_POID           POID [0] 0.0.0.1 /event/session -1 0
0 PIN_FLD_RESULTS        ARRAY [4] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ     POID [0] 0.0.0.1 /account 1 0
```

The output flist might look like this:

```
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_RESULTS       ARRAY [0] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_RESULTS       ARRAY [1] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_RESULTS       ARRAY [2] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_RESULTS       ARRAY [3] allocated 20, used 1
1 PIN_FLD_ACCOUNT_OBJ    POID [0] 0.0.0.1 /account 1 0
```

About Multischema (Global) Searches

BRM includes an alternative set of opcodes designed for use with multiple database schemas. Global searches on multiple schemas are similar to searches on a single schema with two main differences:

- No database number is specified in the search POID.
- The PCM_OP_GLOBAL_SEARCH opcode is called instead of the PCM_OP_SEARCH opcode.

PCM_OP_GLOBAL_SEARCH enables a client application to search for objects that meet a set of criteria defined by the client application. Use this opcode when you do not know enough about the target object to specify its database schema. If you know the specific schema to search, use PCM_OP_SEARCH instead.

Use PCM_OP_SEARCH and the other single-schema search opcodes whenever you can because single-schema searches are the most efficient. A global search is expensive because it is performed synchronously. It opens a context to each schema and waits for all results to be returned before merging the results.

> **Note:**
>
> - Avoid using global searches when transactions are open because they can cause the database to be locked.
> - Single-schema searches are useful only when you know the database number of the schema. If you do not know the specific schema to search, you must use a global search.

For more information, see the following opcodes:

- To perform a global search, use the PCM_OP_GLOBAL_SEARCH opcode. See "Performing a Global Search".
- To perform a global step search, use the following opcodes:
  - PCM_OP_GLOBAL_STEP_SEARCH
  - PCM_OP_GLOBAL_STEP_NEXT
  - PCM_OP_GLOBAL_STEP_END

  See "Performing a Global Step Search".

Performing a Global Search

To perform a global search, use the PCM_OP_GLOBAL_SEARCH opcode. This opcode searches for objects across multiple database schemas.

This opcode enables a client application to search for objects that meet a set of criteria defined by the client application. Use this opcode when you do not know enough about the target object to specify its database schema. If you do know the specific schema to search, use PCM_OP_SEARCH instead.

The input flist contains a search template for an object in the database. The element ID of the PIN_FLD_ARGS element on the input flist specifies which argument is contained on its sub-flist. The maximum number of search arguments is 32.

The output flist is a list of objects that meet the search criteria. An array of PIN_FLD_RESULTS elements is returned, one for each object that was matched.

If one or more schemas are returning errors, those schemas are excluded from the search, and this opcode continues the search across the rest of the schemas. Errors that cause this exclusion include the following:

- A PCP context cannot be opened.
- A socket cannot be opened to a DM.
- A DM connection cannot be set to asynchronous mode.
- A send operation to the DM fails.
- A receive operation from the DM fails.

If the value of the PIN_FLD_RESULTS element on the input flist is **NULL**, each element of the returned array contains all the fields from the matched object.

If the PIN_FLD_RESULTS element on the input flist contains a sub-flist with fields, only those fields specified are returned for each of the matched objects.

You can manipulate the search results using the Flist Field Handling Macros. See "Flist Field-Handling Macros" in *BRM Developer's Reference*.

Limitations

The PCM_OP_GLOBAL_SEARCH opcodes cannot do the following:

- Perform ORDER-BY searches
- Use the **PCM_OPFLG_CALC_ONLY** parameter

Transaction Cache

To improve performance, set the PCM_OPFLG_CACHEABLE flag.

Before the start of any search opcode, the Connection Manager (CM) writes into the database all objects that are in writable cache, have changed during the transaction, and are the same object type as the expected results of the search.

See "Improving Performance when Working with Objects".

Examples

See the sample test program **sample_search.c** in *BRM_home***/apps/sample**. This program includes examples of the following:

- A read object search with a single result expected
- A read fields search with multiple results expected

Performing a Global Step Search

To perform a global step search, use the PCM_OP_GLOBAL_STEP_SEARCH opcode. This opcode step searches for objects across multiple BRM database schemas. This opcode enables a client application to define search criteria, search for objects using that criteria, and receive a specified number of result sets.

> **Note:**
>
> - If you are searching for an object in a known database schema, use PCM_OP_STEP_SEARCH instead.
> - When you perform a global step search, follow best practices and call PCM_OP_GLOBAL_STEP_SEARCH before you call PCM_OP_GLOBAL_STEP_END. Calling PCM_OP_GLOBAL_STEP_END in the incorrect sequence might cause the CM to crash.

The search criteria are passed in by the client application in the form of a PCM_OP_GLOBAL_SEARCH input flist. The input flist contains a search template for an object in the database. The element ID of the PIN_FLD_ARGS element on the input flist specifies which argument is contained on its sub-flist. The maximum number of search arguments is 32.

Be careful when you pass in the search criteria.

- If you have *n* schemas and ask for *m* results, (*n* x *m*) results are fetched. Any extra results are cached in memory in the CM.

- A global search on *n* schemas opens *n* sockets, which could adversely affect performance.

PCM_OP_GLOBAL_STEP_SEARCH only initiates step searching and gets the first set of PIN_FLD_RESULT elements. PCM_OP_STEP_NEXT retrieves the next specified number of results. The PCM_OP_GLOBAL_STEP_NEXT opcode only receives results; it does *not* do a search. PCM_OP_GLOBAL_STEP_END ends the step search, freeing the database cursor and returning any shared memory allocated for the results by the DM.

Stepping backward through the result set is not supported.

No shared memory is allocated in the DM for the results until PCM_OP_GLOBAL_STEP_NEXT gets a part of the result set. When PCM_OP_STEP_END ends the search, the shared memory is freed. An array of PIN_FLD_RESULTS elements is returned, one for each object that was matched.

If the value of the PIN_FLD_RESULTS element on the input flist is **NULL**, each element of the returned array contains all the fields from the matched object.

If the PIN_FLD_RESULTS element on the input flist contains a sub-flist with fields, only the specified fields are returned for each of the matched objects.

The array size of PIN_FLD_RESULTS determines the number of PIN_FLD_RESULT elements to return to the client.

An error is returned to the client if two PCM_OP_GLOBAL_STEP_SEARCH opcodes are sent to the server. If the client is in the middle of a step search, the first search must be ended before another is initiated.

This opcode uses the same input and output flists as PCM_OP_GLOBAL_SEARCH.

Transaction Cache

To improve performance, set the PCM_OPFLG_CACHEABLE flag.

Before the start of any search opcode, the CM writes into the database all objects that are in writable cache, have changed during the transaction, and are the same object type as the expected results of the search.

See "Improving Performance when Working with Objects".

Examples

The *BRM_home*/**apps/sample/sample_search.c** file contains example step searching code.

Getting the Next Set of Search Results from a Global Step Search

To get the next set of search results, use the PCM_OP_GLOBAL_STEP_NEXT opcode.

This opcode enables a client application to receive the next set of results from a search initiated by PCM_OP_GLOBAL_STEP_SEARCH.

The PCM_OP_GLOBAL_STEP_SEARCH opcode determines the criteria for the search, sets the size of the results, and initiates the search. See that opcode for details. This opcode only receives results; it does not perform the search. PCM_OP_GLOBAL_STEP_END ends the step search, freeing the database cursor and returning any shared memory allocated for the results by the DM.

This opcode returns the results of the search in discrete chunks. That is, it goes to the DM and gets the next set of PIN_FLD_RESULT elements. You determine the size of this result set by using the PIN_FLD_RESULTS field on the input flist.

You can manipulate the search results by using the flist field handling macros. See "Flist Field-Handling Macros" in *BRM Developer's Reference*.

This opcode uses the same input and output flists as PCM_OP_GLOBAL_SEARCH.

Transaction Cache

To improve performance, set the PCM_OPFLG_CACHEABLE flag.

Before the start of any search opcode, the CM writes into the database all objects that are in writable cache, have changed during the transaction, and are the same object type as the expected results of the search.

See "Improving Performance when Working with Objects".

Example

The *BRM_home*/**apps/sample/sample_search.c** file contains example step searching code.

Ending a Global Step Search

To end a global step search, use the PCM_OP_GLOBAL_STEP_END opcode. This opcode ends global step searching that has been initiated by PCM_OP_GLOBAL_STEP_SEARCH.

PCM_OP_GLOBAL_STEP_SEARCH sets the criteria for a step search, sets the size of the results, and initiates the search. See that opcode for details. PCM_OP_GLOBAL_STEP_NEXT only receives results; it does not do a search. This opcode ends the step search, freeing the database cursor and returning any shared memory allocated for the results by the DM.

This opcode uses the same input and output flists as PCM_OP_GLOBAL_SEARCH.

The *BRM_home*/**apps/sample/sample_search.c** file contains example step-searching code.

Global Search Example

This example searches for all accounts with a billing cycle of 6 months:

```
/*
* Create the search flist.
*/
s_flistp = PIN_FLIST_CREATE(ebufp);

/*
* Create and add the search poid.
*/
search_poidp = PIN_POID_CREATE((int64)0, "/search", (int64)-1, ebufp);
PIN_FLIST_FLD_PUT(s_flistp, PIN_FLD_POID, (void *)search_poidp, ebufp);

/*
* Add the search template.
*/
PIN_FLIST_FLD_PUT(s_flistp, PIN_FLD_TEMPLATE,
        (void *)"select X from /account where F1 = V1 ", ebufp);

/*
* Add the search argument.
*/
arg_flistp = PIN_FLIST_ELEM_ADD(s_flistp, PIN_FLD_ARGS, 1, ebufp);
num_monthly_cycles = 6;
PIN_FLIST_FLD_PUT(arg_flistp, PIN_FLD_BILL_WHEN,
        (void *)&num_monthly_cycles, ebufp);
```

```
/*
* Add the results we want to fetch.
*/
rslt_flistp = PIN_FLIST_ELEM_ADD(s_flistp, PIN_FLD_RESULTS, 0, ebufp);
PIN_FLIST_FLD_PUT(rslt_flistp, PIN_FLD_POID, (void *)NULL, ebufp);
PIN_FLIST_FLD_PUT(rslt_flistp, PIN_FLD_ACCOUNT_NO, (void *)NULL, ebufp);
PIN_FLIST_FLD_PUT(rslt_flistp, PIN_FLD_STATUS, (void *)NULL, ebufp);

/*
* Do the search.
*/
PCM_OP(ctxp->pcm_ctxp, PCM_OP_GLOBAL_SEARCH, PCM_OPFLG_READ_UNCOMMITTED,
        s_flistp, &r_flistp, ebufp);
```

Building the POID for the Input Flist

With multiple database schemas, all billing information, such as bill items and events for an account, must be in the same schema where the account is located. A common algorithm includes finding the POID for a BRM account object and then searching for additional data that is directly related to that account, such as bill items and events.

When building the PIN_FLD_POID (**FldPoid.getInst()** in Java) for the opcode input flist, it is common to use the database number of the login context. This *does not work* for multiple schemas. Instead, build the PIN_FLD_POID for the search input flist by using the database number from the account POID.

Building POID for the Input Flist in C

```
void FindBillItemsForAnAccount(pcm_context_t* pContext, poid_t* pAcctPoid)
{
// Start building the search input FList.

// Obsolete way:
int64 ContextDB = pin_poid_get_db(pcm_get_userid(pContext));
poid_t* pSearchPoid = PINApp::PoidCreate( ContextDB, _T("/search"), 0, &ebufp );

// Right way:
poid_t* pSearchPoid = PINApp::PoidCreate( pin_poid_get_db(pAcctPoid), _T("/search"), 0,
&ebufp );
.
.
}
```

Building POID for the Input Flist in Java

```
void FindBillItemsForAnAccount(PortalContext connection, Poid acctPoid)
{
// Start building the search input FList.

// Obsolete way:
Poid searchPoid = new Poid( connection.getCurrentDB(), 0, "/search" );

// Right way:
Poid searchPoid = new Poid( acctPoid.getDb(), 0, "/search" );
.
.
}
```

The Impact of Searches on Shared Memory Allocation

For information about the shared memory implications of searching using PCM_OP_SEARCH or PCM_OP_GLOBAL_SEARCH, see "How BRM Allocates Shared Memory for Searches" in *BRM System Administrator's Guide*.

Improving Search Performance

Search operations often constitute most of the activity in the BRM database. When appropriate, it is a good idea to use the following techniques to improve search performance.

Removing Redundant Distinct Searches

Using a distinct search (SRCH_DISTINCT flag) is helpful when a query might return multiple copies of the same object, but it can decrease performance when used unnecessarily. To improve search performance, you can prevent the Oracle DM from using a distinct search when it is redundant.

For example, the following flist search returns **/account** POIDs, which are always unique and cannot contain multiple copies of the same object. Including a distinct search with this example would be redundant.

```
0 PIN_FLD_POID              POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_RESULTS          ARRAY [*] allocated 20, used 2
1     PIN_FLD_POID          POID [0] NULL poid pointer
1     PIN_FLD_ACCOUNT_NO     STR [0] NULL str ptr
0 PIN_FLD_ARGS             ARRAY [1] allocated 20, used 1
1     PIN_FLD_ACCOUNT_NO     STR [0] "%"
0 PIN_FLD_TEMPLATE          STR[0] "select X from /account 1 where lower
(1.F1) like V1"
0 PIN_FLD_FLAGS              INT [0] 256
```

You can configure the Oracle DM to remove or log unnecessary instances of the SRCH_DISTINCT flag when performing searches by setting the following flag in your Oracle DM configuration file (*BRM_home***/sys/dm_oracle/pin.conf**):

-dm audit_search_distinct_flag *value*

where *value* is one of these:

- NO_FLAG (**0**): Do not perform removal and logging. This is the default.
- AUDIT_DISTINCT_WARN_ONLY (**1**): Only log a warning for each unnecessary use of the distinct flag during searches. Removal is not performed.
- AUDIT_DITINCT_WARN_AND_RESOLVE (**2**): Removes any unnecessary use of the distinct flag during searches. It also logs a warning before each removal.
- AUDIT_DISTINCT_RESOLVE_SILENTLY (**3**): Silently removes unnecessary use of the distinct flag during searches without logging any warnings.

For example, this configures the Oracle DM to remove any use of unnecessary distinct flags during searches without logging any warnings:

```
-dm audit_search_distinct_flag 3
```

Step Search Limits

You can improve search performance by limiting the size of search results.

You can specify the maximum number of objects to be returned for the entire step search (that is, for all steps of the search) by specifying the optional PIN_FLD_RESULTS_LIMIT field in the input flist of the following opcodes:

- PCM_OP_SEARCH
- PCM_OP_GLOBAL_SEARCH
- PCM_OP_STEP_SEARCH
- PCM_OP_GLOBAL_STEP_SEARCH

In the PCM_OP_SEARCH and PCM_OP_GLOBAL_SEARCH opcodes, PIN_FLD_RESULTS_LIMIT 100 has the same effect as PIN_FLD_RESULTS [100].

This information helps the RDBMS run the search more efficiently.

> **Note:**
>
> This limit does not apply to any search that uses an **order by** clause. It also does not apply to the PCM_OP_STEP_NEXT or PCM_OP_GLOBAL_STEP_NEXT opcodes.

Limiting the size of search results helps the database to process the query more efficiently because it can stop processing as soon as it has fetched the required number of results. For example, if a search yields 4,000,000 qualifying results, PIN_FLD_RESULTS_LIMIT 100 stops the processing after 100 matching results are found, so only a small subset of the data is scanned.

Transaction Caching

You can improve performance by caching transactions. Without caching, search operations can repeatedly search the same data object within one transaction. By letting the CM cache transactions, you eliminate this redundancy and speed up transaction processing. See "Improving Performance when Working with Objects".

20

Adding Support for a New Service

Learn how to add a custom service to your Oracle Communications Billing and Revenue Management (BRM) system.

Topics in this document:

- About Adding Support for a New Service
- About BRM Services
- Creating Service and Event Storable Classes
- Setting Up Rating for a New Service
- Setting Up Billing for a New Service
- Setting Up Account Creation for a New Service
- Optional Support for a New Service

About Adding Support for a New Service

This document provides an overview of adding a custom service to your BRM system. It provides an end-to-end survey of the tasks you must perform from creating service and event storable classes to setting up rating and billing for a new service. Each task provides a cross-reference to the document that contains more detailed information.

You may need to perform additional tasks not described in this document, depending on your business needs and the type of service you add.

Creating BRM services requires the following:

- Knowledge of programming in C or C++.
- A good understanding of the following BRM components:
  - BRM system architecture.
  - Storable classes and flists (field lists).
  - Opcodes and the Portal Information Network (PIN) library routines.
  - BRM error handling.

About BRM Services

A service is a capability that you provide to customers, such as telephony, broadband access, and email. BRM comes ready to use with a set of preconfigured services, and you can create your own custom services.

About Supporting a New Service

Before adding a new BRM service, you must understand the details of the service and how it will be implemented. For example, if you offer a wireless telephony service, you might want to

track customer logins and provide optional support for add-on services such as text messaging.

To offer a service that is not supported by default in BRM, you may need to perform some of the following tasks to implement the new service:

- Create the storable classes and any custom fields the service requires. See "Creating Service and Event Storable Classes".
- Set up pricing data for the new service. See "Setting Up Rating for a New Service".
- Set up bill items to bill for usage of a new service. See "Setting Up Billing for a New Service".
- Enable accounts to use the new service. See "Setting Up Account Creation for a New Service".

Creating Service and Event Storable Classes

To add a new service, first define your service by determining which information you must track and rate. Then use Storable Class Editor in Developer Center to create new storable classes and any custom fields required to store custom service attributes.

You can add a new service to BRM by creating a subclass of an existing BRM service storable class (for example, **/service/telco/gsm/***service_subclass*) or by creating a new base storable class (**/service/***service_type*).

You may also need to create other storable classes and custom fields to store information about the new service. For example:

- To capture rating information for the service, you might need to create a new **/event** subclass (for example, **/event/session/telco/***service*).
- For offline charging, you must create a corresponding delayed event storable class for any new **/event** storable class you added (for example, **/event/delayed/session/telco/***service*).
- To store custom configurations for a new service, you might need to add fields to or create a subclass of a **/config** storable class. For example:
  - To set up provisioning tags for a new Global System for Mobile Communications (GSM) service, create a new **/config/telco/gsm/***service* storable class.
  - To store service-order state changes for devices associated with a new service, create a **/config/telco/service_order_state/***service* storable class.
  - To allow CSRs to adjust events for a new service, add the event type to the **/config/adjustment/event** object.
- To collect custom profile information for a new service, create a new **/profile** subclass.
- To store failed call records for a new service whose events are suspended by Suspense Manager, you might need to create a **/suspended_usage/***service* storable class. You must create this new subclass only when the fields unique to the new service type are among the queryable or editable fields in suspense.

To create storable classes and custom fields for a new service, perform the tasks in "Creating Custom Fields and Storable Classes".

Use Opcode Workbench to create test instances of your new storable classes.

Setting Up Rating for a New Service

To enable BRM to rate a new service, you must set up pricing data for the new service and configure Rated Event Loader to load rated event data into the database.

Setting Up Pricing Data for Online Rating

You set up service-specific pricing data for online rating by using configuration files. You then add the service to your product offerings by creating charge offers for the new service and adding those charge offers to bundles.

Mapping Event Types to a New Service Storable Class

You map a new service to the events used to rate the service, including any new events you add for the service.

Defining RUMs for New Service Usage Events

You define ratable usage metrics (RUMs) to charge for events. If a new service requires RUMs that are not yet defined, you can define new RUMs.

Setting Up Provisioning Tags for a New Service

You use charge-offer provisioning to rate a service differently based on charge offer attributes. You implement charge-offer provisioning by defining provisioning tags.

You can use provisioning tags to define any kind of attribute. For example, provisioning tags for prepaid services define the following service attributes:

- Extended rating attributes (ERAs)
- Supplementary services for a GSM service
- Bearer services or other service extensions for a telco service

How you define provisioning tags depends on various factors, such as the type of service, whether you must create new ERAs, and whether you must use the provisioning tag with discounts.

Defining Impact Categories for a New Service

You use impact categories to apply different balance impacts for the same charge based on event attributes, such as call origin and destination.

You assign charges to impact categories when you create your pricing components in PDC.

Defining Custom Balance Elements for a New Service

If a new service requires a balance element (such as an aggregation counter balance element) that is not already defined, define the balance element so that BRM can create a balance for it.

You define balance elements in PDC.

Specifying How to Round Balance Impacts for New Service Usage Events

You can configure balance impact rounding for specific types of events. For example, you can round balance impacts of session events differently than you round balance impacts of purchase events. If you do not specify a rounding rule for an event type, BRM uses the default rounding rule.

To specify a rounding rule for an event type you added for a new service, use PDC.

Adding a New Service to Your Product Offerings

After setting up pricing data for a new service, you can create charge offers and add them to your product offerings.

You specify the new service type and event types for the new service when defining various pricing components, such as the following:

- Charge offers:
  - When creating a charge offer.
  - When defining rollover properties if you added a custom noncurrency balance element and you want to enable rollover for that balance element.
- Discounts:
  - When creating a discount.
  - When defining discount exclusion rules.
  - When mapping events to a discount.
- Bundles:
  - When creating a bundle.
  - When making bundles mutually exclusive.
- Packages:
  - When creating a package.
  - When adding bundles to a package.
  - When defining how a package is upgraded or downgraded to another package.
  - When creating a service group.
  - When setting credit limits and thresholds for a subscription service.
  - When setting sub-balance consumption rules for a subscription service.
  - When creating service-level balance groups.
- Charges:
  - When defining a charge.
- Charge selectors:
  - When specifying the service or event fields that determine which charge is selected.

Use PDC to create pricing components that define pricing information for the new service.

Configuring Sub-Balances to Track Specific Types of Usage for a New Service

You can configure BRM keep separate balances for specific types of service usage, such as frequent flyer miles per service instance or minutes per call session.

To keep separate sub-balances for a new service, configure sub-balances for the service's events in the **pin_sub_bal_contributor** file and load the contents of the file into the **/config/sub_bal_contributor** object by running the **load_pin_sub_bal_contributor** utility.

Adding Database Partitions for New Service Usage Events

If you use a partitioned database and you created a new service usage event, add database partitions for the event.

Loading Rated Events for a New Service into the Database

If you rate usage for a new service, you must load rated usage events into the database.

Use Rated Event (RE) Loader to load events for a new service into the BRM database. Set up a processing directory for the new events and configure RE Loader and the RE Loader batch handler to load the events from that directory.

If a new service object includes new fields, you might also need to modify the RE Loader pre-processing script and create new control files.

Setting Up Billing for a New Service

To bill for service usage, configure BRM to pre-create service-level bill items.

To set up service-level bill items, associate the new service type and its usage events with an **/item/misc** object by modifying the **config_item_tags** and **config_item_types** files. Load the contents of these files into the **/config/item_tags** and **/config/item_types** objects by running the **load_config_item_tags** and **load_config_item_types** utilities.

Setting Up Account Creation for a New Service

To successfully create accounts for your new service, you may need to first customize the PCM_OP_CUST_POL_PREP_INHERITED policy opcode to prepare the inherited information for an extended subclass.

You can also write a new policy to implement custom functionality. See "About System and Policy Opcodes".

Setting Up Business Profiles for a New Service

If you use business profiles and you want to specify rules that determine whether a new service meets the requirements of a business profile, modify the **pin_business_profile.xml** file to configure a validation template for the new service type. Load the contents of the file into the **/config/business_profile** object by running the **load_pin_business_profile** utility.

Optional Support for a New Service

This section describes some optional ways you can configure BRM to support a new service.

Synchronizing Data for a New Service with External Applications

Business events capture information needed by external applications or internal BRM components. Business events are used by BRM EAI Manager to ensure data synchronization across applications.

You can add events for a new service to a business event to synchronize any data that the service's event might change. To add an event to a business event, modify the Payload Generator EM configuration file (**payloadconfig.xml**). See "About Publishing Additional Business Events".

For information about EAI Manager, see "About Enterprise Application Integration (EAI) Manager".

Mapping Devices to a New Service

Perform this task if the new service requires a device such as a SIM card or phone number.

To associate a new service with a device, add the service type to your device permit map file (such as **pin_device_permit_map_num**) and load the file into the **/config/device_permit_map** object by running the **load_pin_device_permit_map** utility. See "Defining Device-to-Service Associations".

You can associate services to the following types of devices or to custom device types that you create:

- APNs (access point names)
- IP addresses
- Telephone numbers
- SIM cards
- Vouchers

For information about managing devices and creating new device types, see "Managing Devices with BRM".

Providing Access to a New Service on the Web

Perform this task if you use Self-Care Manager to provide customers with self-care access to services on the Web.

To provide access to information about a new service on the Self-Care Manager home page, customize the Self-Care Manager interface by using the Customer Center SDK. See "Customizing the Self-Care Manager Interface".

Generating Usage Reports for a New Service

If you create a subclass of an existing BRM service storable class, you can run and analyze usage reports for the new service.

21

Using BRM Messaging Services

Learn about the Oracle Communications Billing and Revenue Management (BRM) Universal Message Store (UMS) framework and how to use its components to provide messages on invoices or other documents.

Topics in this document:

- About the UMS Framework
- Enabling Messaging
- Creating and Loading Message Templates
- Generating Messages in the Producer Application
- Retrieving Message Objects in the Consumer Application

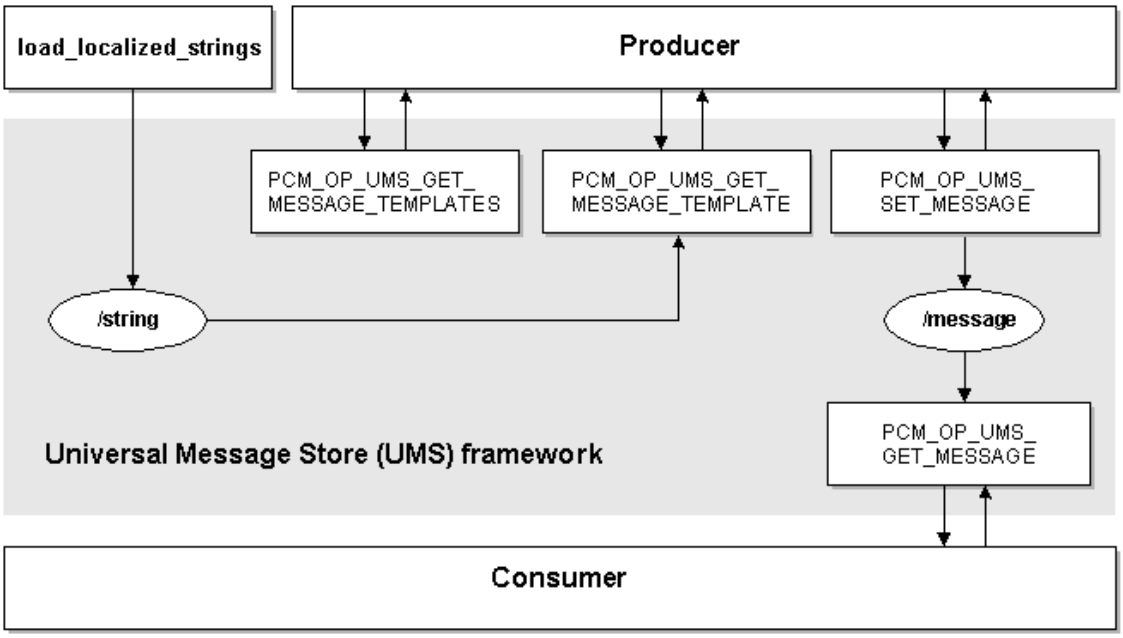
About the UMS Framework

The UMS framework enables you to include system-generated messages in customer documents such as invoices. You can use these messages to market new services or deliver reminders of overdue balances, for example.

UMS works by providing a middle layer between *producers* of messages, such as Collections Manager and third-party customer relationship management (CRM) applications, that generate invoice reminders, and *consumers* of message, such as the invoicing system. This middle layer includes storable classes to store messages and message templates and opcodes that process the messages.

Figure 21-1 illustrates the relationships among the components of UMS. These components are discussed in subsequent sections.

Figure 21-1 UMS Framework Components



To set up messaging, perform these tasks:

- Enabling Messaging
- Generating Messages in the Producer Application
- Retrieving Message Objects in the Consumer Application

Enabling Messaging

To enable messaging, you must modify the PCM_OP_INV_POL_PREP_INVOICE policy opcode. See "About System and Policy Opcodes" for general information about modifying policy opcodes.

To enable messaging:

1. Open **fm_inv_pol_prep_invoice.c**, the source file for PCM_OP_POL_PREP_INVOICE. This file is located in *BRM_SDK_home***/source/sys/fm_inv_pol**.
2. Delete or comment out the following line:

   ```
   #ifdef UMS_MESSAGE_FEATURE
   ```

3. Delete or comment out the **#endif** line following the block of code after the **#ifdef** line.
4. Compile and link **fm_inv_pol_prep_invoice.c** to create a new shared library.
5. Replace the existing shared library on your production system with the new one.
6. Add the following entry as a single line in your CM **pin.conf** to load the required messaging library:

   ```
   - cm fm_module ${PIN_HOME}/lib/fm_ums/${LIBRARYEXTENSION} fm_ums_config -
   pin
   ```

7. Stop and restart the CM.

Creating and Loading Message Templates

Message templates are localizable **/strings** objects in the BRM database. They contain the basic text of a message along with placeholders for specific data such as names, balances, and dates.

You create message templates by writing a localized string file. Each file contains messages for one combination of locale (U.S. English, for example) and domain (invoice reminder messages, for example). The file also contains the text, optionally including placeholders, for one or more message strings. Each string requires a version and ID number.

The name of the template is entered in the HELPSTR field. This name can be displayed in client applications such as Collections Configuration.

The combination of locale, domain, ID, and version must uniquely define each string within the **/strings** storable class.

When you create a message string file, you can include placeholders that are filled with data when the complete **/message** object is created. The placeholder character is a percent sign (%) followed by a number that is incremented for each placeholder. In this example, there are two placeholders:

```
LOCALE = "en_US" ;

DOMAIN = "Messages - invoice reminder" ;
STR
     ID = 0 ;
     VERSION = 1 ;
     STRING = "Your account is now past due in the amount of %1                which was
due on %2. Please send in your payment                promptly." ;
     HELPSTR = "First Reminder";
END
```

The placeholders are replaced with data supplied as elements in the PIN_FLD_ARGS array in the input flist of PCM_OP_UMS_SET_MESSAGE. Element numbers must correspond to placeholder numbers; element 1 replaces **%1**, element 2 replaces **%2**, and so on.

You can include HTML tags in the message string. This is useful when the string will be displayed in an HTML document such as an email or Web page.

You load message templates into the BRM database by running the **load_localized_strings** utility. To overwrite strings with the same version and ID, specify the **-f** option when you run the utility.

Generating Messages in the Producer Application

As their name implies, producer applications supply the messages to the UMS framework. In most cases, a complete message is assembled from a message template that is filled in with data supplied by the producer application.

Retrieving Message Templates

To create a complete **/message** object, the producer application requires the POID of the template on which the message will be based. The first step is gathering a list of available templates. From this list you can extract the POID of the template you want to use.

The producer application calls PCM_OP_UMS_GET_MESSAGE_TEMPLATES to return a list of the POIDs of all message templates for a domain and locale that you specify. For example, if you have stored a number of different marketing message templates for the same locale and domain, the opcode returns a list of the POIDs and template names.

If the producer application includes a graphic user interface (GUI), you can display the available template names for selection by the user. Otherwise, you can select the template Portal object ID (POID) programmatically.

You can also search for the **/strings** object that contains the message you want. This option is particularly useful if you aren't using a GUI and therefore do not need to display the list of available templates. See "Searching for Objects in the BRM Database" for information about searching.

If successful, PCM_OP_UMS_GET_MESSAGE_TEMPLATES returns an array containing the POID and name of each template that matches the locale and domain specified in the input flist.

PCM_OP_UMS_GET_MESSAGE_TEMPLATES stops processing if no templates are available for the specified combination of locale and domain.

Retrieving Message Templates from /strings Objects

To display a full message template in an application, call PCM_OP_UMS_GET_MESSAGE_TEMPLATE. This opcode retrieves the full contents of the template.

If successful, PCM_OP_UMS_GET_MESSAGE_TEMPLATE returns the contents of the specified message template **/strings** object, including the POID, domain, locale, template name, and template string.

PCM_OP_UMS_GET_MESSAGE_TEMPLATE stops processing if the POID of the requested template is incorrect or missing.

You can skip this step if you do not need to display the template contents. The opcode that creates **/message** objects automatically calls PCM_OP_UMS_GET_MESSAGE_TEMPLATE to retrieve the template it needs.

Creating Message Objects

The producer application creates the **/message** object by calling PCM_OP_UMS_SET_MESSAGE_TEMPLATE. The input flist must include the POID of the message template that will be used for this message.

If the template contains placeholders, the input flist must also contain an array whose elements supply data for the placeholder. The data in element 1 replaces placeholder **%1**, the data in element 2 replaces **%2**, and so on.

You also set the scope of the message; whether it applies to a particular bill, or to an account when you create the **/message** object. You define the scope by supplying the POID of the account or bill to which the message applies.

If the message scope is the account, you can also supply an effective date in the input flist.

PCM_OP_UMS_SET_MESSAGE stops processing under these circumstances:

- The POID of the template is missing or incorrect.
- No locale is specified.

- No scoping information (account or bill) is specified.
- No effective date is included for account-scoped messages.

Retrieving Message Objects in the Consumer Application

The consumer application retrieves **/message** objects by calling PCM_OP_UMS_GET_MESSAGE.

The opcode retrieves **/message** objects based on scoping information that you provide. You specify the scope by including the account or bill object POID associated with the messages you want.

By default, PCM_OP_UMS_GET_MESSAGE retrieves all messages that apply at the scoping level you specify. For example, if you specify a **/bill** object, the opcode retrieves all messages that are scoped to that bill and the bill's account with which it is associated. Similarly, if you specify an **/account** object, the opcode retrieves messages scoped to the account.

You can modify the default scoping behavior by including the PIN_FLD_SCOPE field in the input flist with a value of **1**. With this option, PCM_OP_UMS_GET_MESSAGE finds only messages scoped narrowly to the bill or account you specify. For example, including a **/bill** object in the input flist returns only messages specifically scoped to that bill.

You can also include a locale and effective date in the input flist to further narrow the list of messages that is returned.

For each message that matches the scope, PCM_OP_UMS_GET_MESSAGE returns an array that contains the text of the message, the message template name, and the message domain. The consumer application can then select the individual message programmatically or via a GUI.

PCM_OP_UMS_GET_MESSAGE stops processing under these circumstances:

- No scoping information is included in the input flist.
- No **/message** objects exist that meet the scope defined in the input flist.

22

Using BRM with Oracle Application Integration Architecture

Learn how to integrate your Oracle Communications Billing and Revenue Management (BRM) system with external applications, such as financial management software, by using Oracle Application Integration Architecture (Oracle AIA).

Topics in this document:

- About Oracle Application Integration Architecture
- Installing and Configuring the Required BRM Components
- Integrating BRM Features with External CRM Applications
- Integrating BRM Features with External CRM Applications in a Multischema System
- Creating Charge Offers and Discount Offers for an External CRM
- Creating Charge Offers with Different Prices for Multiple Price Lists
- Validating Customer Contact Information

About Oracle Application Integration Architecture

Oracle AIA enables you to set up and orchestrate cross-application business processes so that multiple applications can work together. Oracle AIA runs on top of Oracle Fusion Middleware.

Oracle AIA for Communications pre-built integrations are pre-built packaged process integrations between specific Oracle applications, including Siebel CRM and BRM, based on Oracle AIA. For example, you can use the pre-built integrations to do the following:

- Create charge offers and discount offers in BRM and synchronize them with Siebel CRM, where they can be packaged and purchased by customers.
- Create BRM accounts from accounts created in Siebel CRM.
- Access billing information from BRM to display in Siebel CRM.
- Export general ledger (G/L) data from BRM for import into Oracle Financials.

For more information, see the Application Integration Architecture documentation on Oracle Help Center.

To set up BRM to work with Oracle AIA:

- Install and configure the required BRM components. See "Installing and Configuring the Required BRM Components".
- Integrate BRM features with your external CRM application. See "Integrating BRM Features with External CRM Applications".

Installing and Configuring the Required BRM Components

To integrate BRM with Oracle AIA:

- Install, deploy, and configure JCA Resource Adapter in a J2EE application server. The adapter is the point of connection between BRM and external applications. Requests for information come to the adapter, which then calls BRM opcodes and returns data to the external application.
- Run the **pin_ledger_report** utility to export G/L data to XML files. This data can then be imported into financial management software such as Oracle Financials.

Integrating BRM Features with External CRM Applications

You can integrate the following BRM features with your external CRM applications:

- Collections. See "Integrating Collections with External CRM Applications".
- Friends and family promotions. See "Integrating Friends and Family Promotions with External CRM Applications".
- Invoicing. See "Displaying Siebel CRM Promotion Names on Invoices".

Integrating Collections with External CRM Applications

External CRM applications can track and manage collections activities, such as sending dunning letters, in BRM through the Oracle AIA architecture. Collections data is synchronized between the external CRM application and BRM as follows:

- The external CRM application updates the status of a collections action in BRM by calling the Collections Manager API through Oracle AIA and JCA Resource Adapter.
- Collections Manager notifies the external CRM application when a collections activity occurs, such as an account entering collections, by using the event notification system.
- Oracle AIA retrieves data from Collections Manager by reading views on the BRM collections tables.

To integrate collections with your external CRM:

1. Install the Agent-Assisted Billing Care Process Integration Pack.
2. Configure your external CRM application to send the status of collections actions to the PCM_OP_COLLECTIONS_SET_ACTION_STATUS opcode. The external CRM application sends information to the opcode through Oracle AIA and JCA Resource Adapter.

   The attributes required to call the opcode are listed in the *BRM_home***/apps/brm_integrations/wsdls/BRMCollectionsServices.wsdl** file.
3. Configure BRM to publish **CollectionsAction** business events to the AQ database queue. See "Synchronizing Pricing Data between the BRM Database and External CRMs" in *BRM System Administrator's Guide*.
4. Configure your external CRM application to retrieve the **CollectionsAction** business event from the AQ database queue. See "Synchronizing Pricing Data between the BRM Database and External CRMs" in *BRM System Administrator's Guide*.
5. Configure the BRM **pin_collections_process** utility to publish **CollectionsInfoChange** business events to the AQ database queue:
   a. Open the **pin_collections_process** configuration file (*BRM_home***/apps/pin_collections/pin.conf**) in a text editor.
   b. Add the following entry to the file:

```
- pin_collections_process publish_run_details 1
```

c. Save and close the file.

See "Understanding Collections Manager" in *BRM Collections Manager* for more information.

Integrating Friends and Family Promotions with External CRM Applications

External CRM applications can track and manage friends and family promotions in BRM through the Oracle AIA architecture.

When customers order a friends and family promotion, BRM stores information about the promotion in a provisioning tag. During the pricing data synchronization process, the Oracle DM publishes the provisioning tag and its associated charge offer in a **ProductInfoChange** business event to the AQ database queue. The external CRM application can then retrieve the **ProductInfoChange** business event from the AQ queue and update the information in its system.

To integrate friends and family promotions with your external CRM application:

- Configure BRM to publish **ProductInfoChange** business events to the AQ queue.
- Configure your external CRM application to retrieve the **ProductInfoChange** business event from the AQ queue.

For more information, see "Synchronizing Pricing Data between the BRM Database and External CRMs" in *BRM System Administrator's Guide*.

Displaying Siebel CRM Promotion Names on Invoices

You can set up your system to display Siebel CRM promotion names on customer invoices.

To include Siebel CRM promotion names on BRM invoices:

- Ensure that BRM is configured to display promotion details on invoices.
- Configure Siebel CRM to send information about the promotion to the PCM_OP_SUBSCRIPTION_SET_BUNDLE opcode. The application sends information to the BRM opcode through Oracle AIA and JCA Resource Adapter.

  The attributes required to call the opcode are listed in the *BRM_home***/apps/brm_integrations/wsdls/BRMSubscriptionServices.wsdl** file.
- Use an invoice template that displays promotion names and details.

For more information, see "Adding Siebel CRM Promotion Names to Invoices" in *BRM Designing and Generating Invoices*.

Integrating BRM Features with External CRM Applications in a Multischema System

You can integrate the following BRM features with your external CRM application in a multischema system:

- Account Migration Manager (AMM). See "Integrating Account Migrations with External Applications in a Multischema System" for more information.
- Collections. See "Integrating Collections with External Applications in a Multischema System" for more information.

Integrating Account Migrations with External Applications in a Multischema System

Account migration is synchronized between the external CRM application and BRM in a multischema system as follows:

1. AMM populates the MIGRATED_OBJECTS_T cross-reference table in the primary BRM database with the batch ID and old and new POID values of all the objects that have successfully been migrated.
2. After successfully migrating a group of accounts from one schema to another, BRM generates **AccountInfoChange** business events. These events are sent to the Enterprise Application Integration Data Manager (EAI DM) using an event notification message.
3. The EAI DM publishes the **AccountInfoChange** business events to the Oracle advanced queuing (AQ) database queue.
4. Oracle AIA retrieves the **AccountInfoChange** business events from the AQ database queue and updates the information in the AIA database.
5. Oracle AIA updates the AIA cross-reference table in the AIA database.
6. Oracle AIA deletes the entries in the MIGRATED_OBJECTS_T cross-reference table in the primary BRM database after reading the entries for a particular batch.

To integrate account migration with the external application in a multischema environment:

> **Note:**
>
> Before running AMM, verify that all the collections related Oracle Data Integrator jobs have been completed successfully. For more information on the Oracle Data Integrator jobs, see the Oracle Data Integrator documentation.

1. Install the Oracle AIA for Communications pre-built integration that integrates BRM with your external application. For more information on the Oracle AIA for Communications pre-built integrations, see the Oracle AIA documentation.
2. Add the **publish_migrated_objects** entry to the AMM **Infranet.properties** file by doing the following:
   a. Open the *BRM_home*/**sys/amt/Infranet.properties** file in a text editor.
   b. Add the following entry:

      ```
      publish_migrated_objects = value [, value2 ...]
      ```

      where *value* is a comma-separated list of storable classes whose objects are stored in the MIGRATED_OBJECTS_T cross-reference table.

      > **Note:**
      >
      > If you do not set this entry, AMM will not integrate with an external application in a multischema environment.

c. Save and close the file.

3. Configure BRM to publish **AccountInfoChange** business events to the AQ database queue. See "Configuring the EAI Payload to Synchronize Data" in *BRM System Administrator's Guide*.
4. Configure your external application to retrieve the **AccountInfoChange** business event from the AQ database queue. See "About Retrieving Specific Events from the AQ Database Queue" in *BRM System Administrator's Guide*.

Integrating Collections with External Applications in a Multischema System

To integrate collections with your external application using Oracle AIA in a multischema system, BRM populates the custom views in BRM collections tables with the **/collection_actions** POID schema number, hard-coded as 0.0.0.1 regardless of the schema where the account resides. For **/account** and **/billinfo** (bill unit) objects, the external application reads the schema where the account is residing, whether it is a single-schema or a multischema environment.

See "Integrating Collections with External CRM Applications" for information on integrating collections with your external application.

Creating Charge Offers and Discount Offers for an External CRM

When you create charge offers and discount offers to use in an external CRM, do not add them to bundles and packages. External CRMs, such as Siebel CRM, do not use BRM bundles and packages. They create their own bundles with the BRM charge offers and discount offers.

You use the **pin_export_price** utility to export charge offers and discount offers from BRM to external CRMs.

Creating Charge Offers with Different Prices for Multiple Price Lists

When you use Oracle AIA to integrate BRM with an external application, such as a CRM application, you can configure prices in BRM charge offers to vary based on the price list used by the external application. To do this, when you configure a charge offer in PDC, you create charge selector rules that associate the external application's price list names with the appropriate charges.

When the charge offer is purchased, the price list name is saved with the purchased offer information in BRM. Before the BRM rating engine calculates each charge, the price list name is added to the event being rated, and the rating engine uses that name and the charge selector rules to determine the price.

For information about Siebel CRM price lists, see the following:

https://docs.oracle.com/cd/E58077_01/doc.116/e55959/chap3_mdmsiebelcrm.htm#CHDJHDID

Validating Customer Contact Information

During account creation, BRM validates the format of customer contact information, such as phone numbers, before creating the account in the BRM database. For example, when a customer or customer service representative enters a phone number, BRM validates that the

phone number includes the correct number of digits or contains parentheses around area codes.

By default, BRM accepts nine different telephone number formats. If your external CRM does not use one of the default formats, you must either:

- Add your telephone number format to the list of acceptable formats.
- Configure BRM to accept any telephone number format.

> **Note:**
>
> BRM stores customer contact data in the format passed in by the external CRM. BRM also uses this same format when creating customer invoices. You must configure your external application to pass in phone numbers in the format you would like displayed on customer invoices.

For information about setting the valid telephone number formats, see "Customizing Account Creation" in *BRM Managing Customers*.

23

Using Event Notification

Learn how to enable and use the event notification feature in Oracle Communications Billing and Revenue Management (BRM).

Topics in this document:

- About Event Notification
- Implementing Event Notification
- About Notification Events

About Event Notification

Event notification automatically triggers BRM operations when specified events occur.

The triggering events are mapped to one or more opcodes in a configuration object (**/config/notify**) stored in the BRM database. When any event occurs, BRM checks whether the event is listed as a triggering event in the configuration object. If it is, BRM calls the opcode or opcodes mapped to the event. The information in the event is passed to the opcodes in their input flists. Optionally, a flag can also be passed to the opcodes.

By default, event notification is not enabled. To enable and customize this feature, see "Implementing Event Notification".

About the Event Notification List

The *event notification list* contains all the events that trigger event notification in your BRM system. Each event in the list is mapped to the opcode or opcodes that are run when the event occurs. The event notification list is stored in the **/config/notify** object in your BRM database.

By default, the event notification list is not loaded into the database. To load the list into the database, you must first set up the list in a configuration file. Depending on which BRM features you use, your system may contain one or more of the following configuration files for event notification. Each file contains default event-to-opcode mapping that supports event notification for one or more BRM features. All of the event notification configuration files available in your system are in the *BRM_home***/sys/data/config** directory, where *BRM_home* is the directory in which the BRM server software is installed.

- **pin_notify**: Supports the following features:
  - Automated Monitor Setup (AMS)
  - Device management
  - Discounting
  - Email notification
  - Event rerating
  - Midcycle charge offer charge-change calculations
  - Balance reservation for disputes and settlements

- **pin_notify_eai**: Supports Enterprise Application Integration (EAI) Manager.
- **pin_notify_ifw_sync**: Supports account synchronization and Suspense Manager.
- **pin_notify_ipc**: Supports policy-driven charging.
- **pin_notify_kafka_sync**: Supports the Kafka DM.
- **pin_notify_ldap**: Supports Lightweight Directory Access Protocol (LDAP) Manager.
- **pin_notify_plugin_http**: Supports the EAI Manager **dm_http** plug-in. See "Configuring EAI Manager to Publish to an HTTP Port".
- **pin_notify_ra**: Supports Revenue Assurance Manager and Suspense Manager.
- **pin_notify_telco**: Supports Global System for Mobile Communications (GSM) Manager.

To modify the content of one of these files, see "Editing the Event Notification List".

If your system contains more than one of these files, you must merge their contents into a single file. See "Merging Event Notification Lists".

To load the content of one of these files into the BRM database, see "Loading the Event Notification List".

> **Note:**
>
> Configuring notification thresholds that result in large number of subscriber breaches can impact call detail record (CDR) throughput.

Implementing Event Notification

By default, event notification is not enabled in BRM because the **/config/notify** object is not created during installation. To implement event notification:

1. If your system has multiple configuration files for event notification, merge them. See "Merging Event Notification Lists".
2. (Optional) To accommodate your business needs, add events to or comment them out of the configuration file that contains the final event notification list you want to load into the BRM database. See "Editing the Event Notification List".
3. (Optional) If necessary to accommodate your business needs, create custom code for event notification to trigger. See "Triggering Custom Operations".
4. Load your final event notification list into the BRM database. See "Loading the Event Notification List".

You can customize event notification by using the PCM_OP_ACT_POL_EVENT_NOTIFY opcode. This opcode is called by various event notification processes and by the PCM_OP_ACT_POL_EVENT_LIMIT policy opcode. This opcode processes events for LDAP integration and email notification when invoked by event notification.

Merging Event Notification Lists

To enable event notification, you run the **load_pin_notify** utility to load the configuration file containing your event notification list into the BRM database. *Before* running the utility, however, you must merge configuration files for event notification if either of the following is true:

- You are enabling event notification for the first time, and your system has multiple configuration files for event notification.
- Your BRM database already contains an event notification list, and you want to add an event notification list for another feature to the database.

> **Caution:**
>
> If you load the new feature's list before merging it with your system's current list, you will disable event notification for the features supported by the current list.

To merge event notification lists:

1. In a text editor, open all the event notification configuration files that you want to merge. By default, the files are in the *BRM_home***/sys/data/config** directory.
2. Copy all the entries from the open files into one of the default files or into a new file.

> **Tip:**
>
> Save a copy of the default files before merging them.

3. Save and close the merged file.

> **Tip:**
>
> You can give the merged file any name you want, and you can store it in any location.

To edit the merged file, see "Editing the Event Notification List".

To load the merged file, see "Loading the Event Notification List".

Editing the Event Notification List

Your system's event notification list is set up in a configuration file (see "About the Event Notification List"). To modify the event notification list:

1. In a text editor, open the configuration file that contains the list. By default, the file is in the *BRM_home***/sys/data/config** directory.
2. To add an entry to the list, use this syntax:

```
opcode_number     flag     event
```

where:

- *opcode_number* is the number associated with the opcode run when the event occurs. Opcode numbers are defined in header (***.h**) files in the *BRM_home***/include/ops** directory.
- *flag* is the name of the flag to pass to the opcode when it is called by the event notification feature. **0** means no flag is passed.

- *event* is the name of the event that triggers the execution of the opcode. You can use any BRM default or custom event defined in your system. Triggering events do not have to be persistent. For example, you can use notification events (see "About Notification Events") and events that you have excluded from the BRM database (see "Managing Database Usage" in *BRM System Administrator's Guide*).

For example:

```
301     0     /event/session
```

This example specifies that when an **/event/session** event occurs, the event notification feature calls opcode number 301, which is the PCM_OP_ACT_POL_EVENT_NOTIFY policy opcode, passing it the contents of the event but not passing it any flag.

To run multiple opcodes when an event occurs, see "Triggering Multiple Opcodes with One Event".

3. To disable an entry in the list, insert a number sign (#) at the beginning of the entry. For example:

```
# 301     0     /event/session
```

4. Close and save the edited file.

> **Tip:**
>
> You can give the file any name you want, and you can store it in any location.

5. Load the edited list into the BRM database. See "Loading the Event Notification List".

Triggering Multiple Opcodes with One Event

If an event is mapped to more than one opcode in the event notification list, BRM runs the opcodes in the order they are listed whenever the event is generated.

The following example specifies that when an **/event/provisioning/service_order/telco/gsm** event is generated, BRM first runs PCM_OP_TCF_PROV_HANDLE_SVC_ORDER (opcode number 4017), then runs PCM_OP_TCF_PROV_UPDATE_PROV_OBJECT (opcode number 4019).

```
4017   0   /event/provisioning/service_order/telco/gsm
4019   0   /event/provisioning/service_order/telco/gsm
```

Triggering Custom Operations

To use event notification to trigger custom operations not included in an existing policy opcode:

1. Add the operation to the PCM_OP_ACT_POL_EVENT_NOTIFY policy opcode.
2. Add the following entry to your system's event notification list:

```
301     flag     event
```

For more information, see "Editing the Event Notification List".

> **Note:**
>
> 301 is the number of PCM_OP_ACT_POL_EVENT_NOTIFY.

Loading the Event Notification List

To add your event notification list to the BRM database, run the **load_pin_notify** utility. The utility loads the list into the **/config/notify** object.

> **Caution:**
>
> This utility replaces the current list in the **/config/notify** object with the list in the configuration file that you load. If you use event notification for multiple features, you must merge the old list with the new list *before* running this utility. Otherwise, you will lose existing event notification functionality. See "Merging Event Notification Lists".

> **Note:**
>
> To connect to the BRM database, this utility needs a configuration (**pin.conf**) file in the directory from which you run the utility. For information about creating configuration files for BRM utilities, see "Creating Configuration Files for BRM Utilities" in *BRM System Administrator's Guide*.

To load the event notification list:

1. Go to the directory that contains the list you want to load. By default, configuration files containing event notification lists are in the *BRM_home***/sys/data/config** directory.
2. If necessary, do one or both of the following:
   - Edit the list. See "Editing the Event Notification List".
   - Merge the list with other event notification lists. See "Merging Event Notification Lists".

   > **Caution:**
   >
   > This utility overwrites all existing data in your system's **/config/notify** object. If you are updating the event notification list, you cannot load new or changed entries only. You must load the entire list each time you run the utility.
3. If you edited or merged the list, save the configuration file that contains the final list.

> **Note:**
>
> You can give the file any name you want, and you can place the file anywhere you want.

4. Use the following command to run the **load_pin_notify** utility:

   ```
   load_pin_notify event_notification_configuration_file_name
   ```

   If you do not run the utility from the directory in which the configuration file is located, include the complete path to the file. For example:

   ```
   load_pin_notify BRM_home/sys/data/config/event_notification_configuration_file_name
   ```

5. Stop and restart the Connection Manager (CM).
6. (**EAI Manager only**) Stop and restart the EAI Data Manager (DM):

   ```
   cd BRM_home/bin
   stop_dm_eai
   start_dm_eai
   ```

7. (**EAI Manager only**) Stop and restart the Payload Generator External Module (EM):

   ```
   cd BRM_home/bin
   stop_eai_js
   start_eai_js
   ```

8. (**GSM Manager only**) Start the Provisioning DM:

   ```
   start dm_prov_telco
   ```

9. To verify that the event notification list was loaded, use one of these features to display the **config/notify** object:
   - Object Browser
   - **robj** command with the **testnap** utility

   > **Note:**
   >
   > By default, the BRM database does not contain the **/config/notify** object. The object is created when you run the **load_pin_notify** utility.

   For information about reading an object and writing its contents to a file, see "Reading an Object and Writing Its Contents to a File".

About Notification Events

Any subclass of the **/event** storable class can be used to trigger event notification. When standard **/event** subclasses are used to trigger event notification, the information their instances contain is handled as follows:

- It is added to the input flist of the opcode or opcodes that are run.
- It is stored in the BRM database.

For more information about standard **/event** subclasses, see *BRM Storable Class Reference*.

Unlike instances of standard events, instances of **/event/notification** subclasses (*notification events*) are not persistent. Hence, the information they contain is not stored in any database. Instead, it is used only to populate the input flists of opcodes run by the event notification feature.

24

Writing Custom Batch Handlers

Learn about the batch handler feature in Oracle Communications Billing and Revenue Management (BRM) and how to write custom batch handlers.

Topics in this document:

- About Batch Handlers
- Configuration Parameters
- Batch Handler Work Flow

About Batch Handlers

Batch handlers are typically used to launch specific applications on a timed or occurrence-driven basis. Each batch handler can be any executable program or script that can be run from a command line. It can be written in Perl, shell script, C, Java, or any other language, so long as it can call an application and update the status and other fields in the BRM database.

Batch handlers run under the control of the Batch Controller. The Batch Controller lets you specify when to run programs or scripts automatically, either at timed intervals or upon creation of certain files, such as log files. For more information about the Batch Controller, see "Controlling Batch Operations" in *BRM System Administrator's Guide*.

Configuration Parameters

For BRM-related parameters, each batch handler must use a standard **pin.conf** or **Infranet.properties** configuration file. For any other parameters, it can also use its own, separate, configuration file.

You can configure the Batch Controller's *handler_name***.start.string** parameter to pass parameters to your batch handler, using any options except **–p** and **–d**. The command that the Batch Controller issues is actually:

*handler_name***.start.string –p** *handler_poid* **–d** *failed_handler_poid*

Therefore, the **–p** and **–d** options are reserved.

For more about configuration, see "Controlling Batch Operations" in *BRM System Administrator's Guide*.

Batch Handler Work Flow

The Batch Controller uses status values, in the BRM database, to monitor the operation status of batch handlers. Handlers must be carefully coded to set this status to the right values at the right times.

As the handler proceeds, the BRM database keeps track of its status, according to Table 24-1:

Table 24-1 Status Values

Status Keyword	Value
NOT_STARTED	–1
STARTING	0
STARTED	1
INTERRUPTED	2
COMPLETED	19
FAILED_TO_START	50
FAILED_TO_COMPLETE	51

You can define other status values for the handler to set, for tracking its own internal status, but these custom status values must be 100 or greater.

> **Note:**
>
> All values 99 and lower are reserved for BRM.

All values in this table, other than **STARTED** and **COMPLETED**, are set by the Batch Controller. A handler must never set the status of its handler object to any value other than **STARTED**, **COMPLETED**, or a value greater than 99. The controller relies on this value.

When the Batch Controller detects a file or a time that is to trigger a batch handler, it creates a new object in the BRM database to record that occurrence. It then creates new handler objects in the database, with their status set to **NOT_STARTED**.

The Batch Controller then checks to see if it can run a handler. Its configuration specifies a maximum number of handler instances for high-load times and a different maximum for low-load times.

If the applicable maximum has not been reached, the Batch Controller:

1. Sets the handler's status to **STARTING**.
2. Extracts the Portal object ID (POID) of the related handler object.
3. Issues this command:

   handler_name.**start.string –p** *handler_poid*

   to start the handler. The start string can include parameter options other than **–p** and **–d**.

Once the batch handler has started, it must:

1. Read its configuration file or files and configure itself.
2. Connect to the BRM database.
3. Find the corresponding handler object in the BRM database, using the POID that was passed with the **-p** option when the Batch Controller started the handler.
4. Set the status field to **STARTED**. The Batch Controller uses this field to monitor the status of running handlers.

If a handler object continues to show **STARTING** status after a waiting time specified in the Batch Controller's **Infranet.properties** file, the Batch Controller changes the status to **FAILED_TO_START**, and then issues this command to start a replacement copy of the handler:

*handler_name***.start.string –p** *handler_poid* **–d** *failed_handler_poid*

The replacement handler gets its own POID in the **–p** option. It also gets the POID of the failed handler, in the **–d** option.

5. Perform its designed activity, which typically includes starting a BRM feature or other application.
6. Collect the return value of the application that the handler called, and change the status field accordingly:
   a. If the activity has completed satisfactorily, set the status field to **COMPLETED**.
   b. If the return value indicates that the activity was not successful, the handler must set the status field a value greater than 99. After the configured waiting time, if the related handler object does not show status either **COMPLETED** or greater than 99, the Batch Controller changes the status to **FAILED_TO_COMPLETE**. It does *not*, however, start a replacement copy of the handler.

   > **Note:**
   >
   > It is up to you to monitor the BRM database and the Batch Controller's log file, to see if any handler has failed to complete. If a handler does take longer than the timeout period to complete, for any reason, it can still update the status of its handler object to a value greater than 99; to satisfy the Batch Controller's timeout watcher.

7. End the handler's event session.
8. Exit.

When the Batch Controller detects the **COMPLETED** status, it decrements its count of currently running handlers.

25

Managing Devices with BRM

Learn about the Device Management framework in Oracle Communications Billing and Revenue Management (BRM) and how to use its components to build custom device management systems.

Topics in this document:

- About the Device Management Framework
- Implementing Device Management
- Device Management and Multischema Environments
- Configuring Event Notification for Device Management
- Defining the Device Life Cycle
- Defining Device-to-Service Associations
- Creating Custom Device Management Systems

About the Device Management Framework

The Device Management framework enables you to write applications to manage devices in BRM or to connect an existing device management system to BRM. The Device Management framework is used by BRM optional components such as SIM (Subscriber Identity Module) Manager and Number Manager to facilitate their device management features.

The Device Management framework is part of Inventory Manager, which is an optional, separately purchased feature.

In BRM, a *device* can be physical, such as a set-top box, or "virtual," such as a phone number. Both types of devices are represented in BRM as objects.

The Device Management framework provides a storable class, **/device**, which can be subclassed to accommodate specific device types. Each **/device** or its **/device** subclass represents an individual device.

Device management involves creating device objects, associating them with services and accounts, controlling their life cycles, and removing them when they are no longer needed. The framework provides storable classes, standard and policy opcodes, and utilities for these purposes.

Implementing Device Management

The Device Management framework includes opcodes and objects that are used to store and manage devices. It provides the following functionality:

- Creating Devices
- Managing the Device Life Cycle
- Managing Device Attributes
- Associating Devices and Services

- Deleting Devices
- Tracking Device History

Creating Devices

Each device that you track in BRM is represented by a **/device** object. The way these objects are created depends on your business. For example, if you supply the devices yourself, you might create **/device** objects in bulk with a custom-designed application. When a customer is assigned a device, you would use one of the already existing **/device** objects.

On the other hand, if your business works with devices such as SIM cards that customers supply themselves, you might use a custom graphic user interface (GUI) tool to create **/device** objects at the same time that you set up customer accounts.

While there are many scenarios for device creation, the actual process used is always the same: all device-creation applications call PCM_OP_DEVICE_CREATE. This opcode creates **/device** objects using the device type, initial state, and attributes specified in the input flist. See "Managing the Device Life Cycle" for more information about device states and "Managing Device Attributes" for more information about device attributes.

Because your business may need to manage several types of devices, the **/device** storable class can be subclassed to represent *device types*. Each device type is a group of devices that have similar characteristics. For example, the BRM Number Manager creates **/device/num** objects while the SIM Manager creates **/device/sim** objects.

You use Storable Class Editor, part of Developer Center, to create new subclasses. See "Creating Custom Fields and Storable Classes".

A policy opcode, PCM_OP_DEVICE_POL_CREATE, enables you to customize the device creation process. For example, you can customize the policy to ensure that all devices have unique numbers.

Managing the Device Life Cycle

Devices pass through various stages in their life cycle. For example, a set-top box moves from manufacturing to a service provider's inventory, where it might be pre-provisioned. From there it is assigned to a retailer or perhaps directly to a subscriber. After it reaches the end-user, it could be returned for repair, obsoleted, or stolen.

A device management system must keep track of these stages or *device states*. In BRM, the device state is represented by the PIN_FLD_DEVICE_STATE field in a **/device** object. The value of this field represents the current state of the device. The set of possible device states and state transitions defines the *life cycle* of a **/device** object.

Device life cycles are regulated by **/config/device_state** objects in the database. You define device states and state changes in a configuration file and load it into the database by using the **load_pin_device_state** utility. See "Defining the Device Life Cycle" for more information.

Device management applications call the PCM_OP_DEVICE_UPDATE opcode to change device states (or PCM_OP_DEVICE_SET_STATE for state changes only). The opcode validates the change specified in the opcode against the **/config/device_state** object for that device type.

There are two ways to add validation checks or other business logic to a device state change. One way is to modify the PCM_OP_DEVICE_POL_SET_STATE policy opcode. This opcode is called automatically by PCM_OP_DEVICE_SET_STATE before changing the device state. Another way is to add additional validation checks to the policy opcode of your choice or a custom policy opcode, and reference the opcode in the **/config/device_state** object. You do

this by adding opcode numbers to the configuration file that you create for each device, and loading the data in the file into **/config/device_state** by using the **load_pin_device_state** utility. BRM invokes the opcodes specified for each device type. For more information, see "Customizing Device State Changes" and "load_pin_device_state".

Managing Device Attributes

Devices have attributes such as descriptions, serial numbers, and manufacturer names that a device management system needs to track the devices. These attributes can be used to identify devices in GUI device management applications.

Device attributes are stored in **/device** object fields. Attributes applicable to all devices (device ID, description, manufacturer, and model) appear in the base **/device** storable class. You can add fields to **/device** subclasses for attributes specific to particular device types. For example, **/device/sim** contains attributes specific to SIM cards.

Applications call PCM_OP_DEVICE_UPDATE (or PCM_OP_DEVICE_SET_ATTR individually) to set device attributes. This opcode can be used to change the common attributes mentioned above and the attributes introduced in subclasses. It cannot be used to change the device state or service association.

Use the PCM_OP_DEVICE_POL_SET_ATTR opcode to customize setting device attributes. See *BRM Opcode Guide*.

Associating Devices and Services

Devices are associated with services. For example, a set-top box requires movies or other content to be useful. The reverse is also true; a video-on-demand service requires a device to display it.

In BRM, these device-to-service associations are represented by the association of **/device** objects with **/service** objects. For example, when a CSR provisions an account with a service that requires a device, the POID of the **/service** object is added to the PIN_FLD_SERVICES array in the **/device** object. The **/account** object POID associated with the service is also added to the **/device** object.

These device-to-service associations can take place on a many-to-many basis. One device may require multiple services and one service may require multiple devices. For example, a SIM card can be provisioned with several different wireless services.

The **/config/device_permit_map** object determine which service types can be associated with which device types. You define the valid device-to-service mappings in a configuration file and load it into the **/config/device_permit_map** object by running the **load_pin_device_permit_map** utility. See "Defining Device-to-Service Associations".

Applications call PCM_OP_DEVICE_ASSOCIATE to associate and disassociate devices and services for a particular account or at the account level. A flag in the input flist determines whether the services and devices are associated or disassociated. This opcode validates the associations against the **/config/device_permit_map** object for the device type specified in the input flist. Invalid associations cause the opcode to fail.

Deleting Devices

Devices can be removed from service for a variety of reasons. They might be returned by the customer, lost, stolen, or become obsolete. You can call PCM_OP_DEVICE_DELETE to remove the device object from the database when it is no longer needed.

Before you delete a device object, you should disassociate services from the device. During device deletion, the PCM_OP_DEVICE_POL_DELETE opcode checks for services associated with the device. If associations are found, the opcode generates an error and cancels the transaction. If desired, you can modify the policy to prevent the association check or to automatically remove associations before the device object is deleted.

In some cases you may want to replace a device object while keeping service associations intact. In this case, disassociate the services from the old device object and reassociate them with the new device in the same transaction. After the services are reassociated, you can delete the original device object.

See "Associating Devices and Services" for more information about device-to-service associations.

Tracking Device History

All operations to a **/device** object, such as device creation, state transitions, and changes to attributes, are recorded in the database as events. Recording events allows you to track the history of a device.

You can turn off event recording and turn on auditing to maintain a device history without storing details. For information about choosing which events to record, see "Managing Database Usage" in *BRM System Administrator's Guide*. For an introduction to auditing and links to more detailed information, see "About Maintaining an Audit Trail of BRM Activity" in *BRM Managing Customers*.

Two BRM reports enable you to monitor the status and device history of devices. See "About BRM Reports" in *BRM Reports*.

Device Management and Multischema Environments

The Device Management framework supports multischema environments, with some restrictions:

- A **/device** object can be associated only with **/account** and **/service** objects located in the same database schema as the **/device** object.
- **/device** objects are not available across schemas. Device management **/config** objects (**/config/device_state** and **/config/device_permit_map**) are available, however.
- You cannot move **/device** objects from one schema to another. You can delete the objects and re-create them in another schema, but service and account associations are lost.
- You can enforce device number uniqueness across schemas, but this requires the use of global search opcodes and may result in slow performance.

Configuring Event Notification for Device Management

When a device state changes, Device Management uses event notification to call opcodes that perform the appropriate follow-up operations.

Although any subclass of the **/event** storable class can be used to trigger event notification (see "About Notification Events"), Device Management generates **/event/notification/device/state** specifically to use for event notification.

Before you can use Device Management, you must configure the event notification feature as follows:

1. If your system has multiple configuration files for event notification, merge them. See "Merging Event Notification Lists".
2. Ensure that the merged file includes the following information from the *BRM_home*/**sys/data/config/pin_notify** file, where *BRM_home* is the directory in which the BRM server software is installed:

```
# Device Management Framework related event notification
2706    0    /event/notification/device/state
```

3. (Optional) If necessary to accommodate your business needs, add, modify, or delete entries in your final event notification list. See "Editing the Event Notification List".
4. (Optional) If necessary to accommodate your business needs, create custom code for event notification to trigger. See "Triggering Custom Operations".
5. Load your final event notification list into the BRM database. See "Loading the Event Notification List".

For more information, see "Using Event Notification".

Defining the Device Life Cycle

You define the life cycle for device objects in BRM by editing or creating a configuration file and then loading the data into a **/config/device_state** object in the database.

Each **/config/device_state** object contains definitions of the possible device states and state changes for one device type. **/config/device_state** objects also contain information about which policy opcodes to call during state changes.

See "load_pin_device_state" for detailed information about the syntax of the configuration file and running the load utility. See "Customizing Device State Changes" for information about calling policy opcodes.

> **Note:**
>
> You must load the life cycle definitions into the database *before* using any device management features. Because device management configuration data is always customized, no default values are loaded during BRM installation.

While the life cycle for every **/device** object is unique, the general pattern for all **/device** objects is the same:

- During device creation, a **/device** object moves from **Raw** state (state 0) to an initial state that is defined for that device type. **/device** objects cannot be saved in **Raw** state.
- During the life of the **/device** object, it moves from state to state in ways that are defined in a **/config/device_state** object. For example, a device might be able to move from a **Pre-provisioned** state to an **Assigned to Dealer** state, but not directly from **Assigned to Subscriber** state. Each device type has its own **/config/device_state** object, so you can define different life cycles to meet your business needs.
- At the end of the working life of the **/device** object, it moves into a final state such as **Stolen** or **Obsolete**. Depending on business needs, the object could then be deleted to save space.

There can be any number of device states, each of which is assigned a number in the device state configuration file. State 0 is reserved for **Raw** state, but other numbers can be freely assigned.

There are four *state types* that correspond to the stages in the general pattern described above. These state types are defined in the **pin_device.h** header file and cannot be changed.

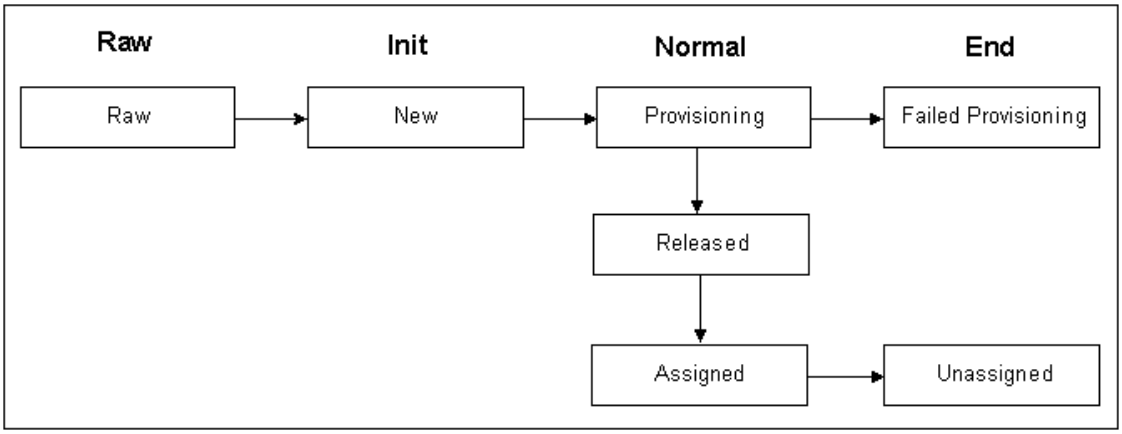
When you define device states, they must be assigned to one of these types:

- **Raw**: There can be only one device state of this type for each device type. **/device** objects are in raw state only during the creation process. They can never be saved in a raw state. Raw states can transition only to Init states.
- **Init**: This state type is for states in which the object can first be saved to the database. There can be more than one initial state for a particular device type. For example, you might want the device to be initialized to one state when created by a batch device creation tool and to another state when created by a CSR using a GUI tool. Init states can transition to Init, Normal, or End states.
- **Normal**: Normal device states are all those that occur between the object's initial state and its end state. You can think of these as the "working" states for the device. There can be any number of Normal device states. Normal states can transition to Init states, other Normal states, and End states.
- **End**: Devices cannot transition from End states. Because there are many possible end-of-life scenarios, you can include any number of End states.

When you define a device life cycle, be sure to consider all the stages in the life cycle of the actual device and all the possible relationships between them. You should also consider the level of detail that is useful to track in your device management system.

Figure 25-1 shows a relatively simple device life cycle for wireless phone SIM cards. The life cycles for your devices may be more complex depending on the characteristics of the devices and your business needs.

Figure 25-1 Wireless Phone SIM Card Life Cycle



Localizing Device State Names

The names you use for device states can be localized for display in a GUI application. You must define the state names as text strings and load the definitions into the database. To localize the device state names, you edit a copy of the **device_states.en_US** sample file in the *BRM_home***/sys/msgs/devicestates** directory and save the edited version with the correct

locale file extension. You then use the **load_localized_strings** utility to load the contents of the file into the **/strings** objects.

When you run the **load_localized_strings** utility, use this command:

```
load_localized_strings device_states.locale
```

> **Note:**
>
> If you're loading a localized version of this file, use the correct file extension for your locale. For a list of file extensions, see "Locale Names".

For information on loading the **device_states.***locale* file, see "Loading Localized or Customized Strings". For information on creating new strings for this file, see "Creating New Strings and Customizing Existing Strings".

Customizing Device State Changes

PCM_OP_DEVICE_UPDATE is used to change device states. It calls PCM_OP_DEVICE_SET_STATE to actually make the change. During the processing of PCM_OP_DEVICE_SET_STATE opcode, there are two opportunities to call policy opcodes:

- The first policy call takes place during the state change itself, just before the transaction is committed to the database.
- The second occurs just after the transaction that changes the device state.

You can use these two policy calls for different purposes. For example, you might want to customize the process for assigning a SIM card to a customer, which involves a state change. During the first policy call by PCM_OP_DEVICE_SET_STATE, the policy opcode could check the customer's handset to ensure compatibility with the SIM card. If the two devices are compatible, the state change takes place. In the second policy call, after the state change transaction is complete, the policy opcode could provision the SIM card by calling PCM_OP_DEVICE_ASSOCIATE.

PCM_OP_DEVICE_POL_SET_STATE is the default policy opcode for device state changes. You can create any number of custom policy opcodes to replace or supplement it. You can also specify additional policy opcodes to call for each state change in the device-specific configuration file that defines the device life cycle. For each state change, you can specify an opcode for one, both, or neither of the two potential policy calls.

For information on naming and setting up this configuration file, see **load_pin_device_state**. For information about the state change opcodes, see:

- PCM_OP_DEVICE_UPDATE
- PCM_OP_DEVICE_SET_STATE
- PCM_OP_DEVICE_POL_SET_STATE

Defining Device-to-Service Associations

You define which device and service types can be associated by editing or creating a configuration file and then loading the data into a **/config/device_permit_map** object in the database.

See **load_pin_device_permit_map** for detailed information about running the load utility and about the syntax of the configuration file. See PCM_OP_DEVICE_ASSOCIATE and PCM_OP_DEVICE_POL_ASSOCIATE for more information about the device-to-service mapping opcodes.

> **Note:**
>
> You must load the mapping information into the database *before* using any device management features. Because device management configuration data is always customized, default values are not loaded during BRM installation.

Use the PCM_OP_DEVICE_POL_ASSOCIATE opcode to customize device association.

Device and service types can be associated in any combination. One device can have any number of associated services. Likewise, one service may require multiple devices.

Creating Custom Device Management Systems

You can create custom device management systems using the Device Management framework. A device management system could be a separate, stand-alone application that performs all necessary tasks, or it could be an enabling application that connects an existing device management system.

A device management application is no different from any other BRM application except that it makes use of the opcodes and storable classes provided by the framework.

"Adding New Client Applications" provides a general overview and information about writing applications in C. Other chapters provide information about additional programming languages.

You will need to complete some or all of the following tasks to implement a device management system:

1. Create a subclass of the **/device** storable class for every new device type. Each subclass should include fields for the unique attributes of the device type.

   You use Storable Class Editor to create fields and storable classes. See "Creating Custom Fields and Storable Classes".

2. If necessary, write a new FM for any unique functionality required by your device management system.

   See "Writing a Custom Facilities Module".

3. If necessary, modify the Device FM policy opcodes to customize the default device management functionality.

   See "About System and Policy Opcodes" for general information and "Device FM Policy Opcodes" in *BRM Opcode Guide* for specific information about the Device policy opcodes.

4. Write a custom application that calls the Device standard and policy opcodes and any custom opcodes you have created.

   See "Adding New Client Applications" for general information and information about writing applications in C.

5. Define the life cycles and device-to-service mappings for your devices.

   See "Defining the Device Life Cycle" and "Defining Device-to-Service Associations".

26

Managing Orders

Learn how to use Oracle Communications Billing and Revenue Management (BRM) Order Manager and its components to build custom order management systems.

Topics in this document:

- About Order Manager
- Implementing Order Manager
- Installing Order Manager
- Order Management and Multischema Environments
- About Defining the Order Life Cycle
- Creating Custom Order Management Systems
- About the Order Management Opcodes

About Order Manager

Order Manager enables you to write applications to manage orders and sub-orders in BRM or to connect an existing order management system to BRM.

Order Manager is part of Inventory Manager, which is an optional, separately purchased feature.

In BRM, an *order* is a request for a physical or virtual object such as a SIM card. An order is represented in BRM as an object. You use Order Manager to create orders for device inventory such as SIM cards. Order Manager provides the **/order** storable class, which you can subclass to represent specific order types. Each object in **/order** or its subclasses represents an individual order.

Order management includes creating the order objects, controlling their life cycles and their attributes, processing orders, tracking their history, and removing orders when they are no longer needed. The framework provides storable classes, standard and policy events, and utilities for these purposes.

Implementing Order Manager

Order Manager provides the following functionality:

- Creating Orders
- Processing Order Response Files
- Managing the Order Life Cycle
- Associating or Disassociating Orders with Master Orders
- Managing Order Attributes
- Deleting Orders
- Tracking Order History

- Managing the Order History Log

Creating Orders

Each order that you track in BRM is represented by an **/order** object. The way these objects are created depends on your business. You can create orders and sub-orders, and you can associate sub-orders with master orders.

While there are many scenarios for order creation, the actual process used is always the same: all order-creation applications call PCM_OP_ORDER_CREATE. This opcode creates BRM **/order** objects that use the order type, initial state, and attributes specified in the input flist. See "Creating /order Objects".

A policy opcode, PCM_OP_ORDER_POL_CREATE, enables you to customize the order creation process.

Because your business may need to manage several types of orders, the **/order** storable class can be subclassed to represent different order types.

You use Storable Class Editor, an application in Developer Center, to create new subclasses. See "Creating Custom Fields and Storable Classes".

Processing Order Response Files

After you create an order, you send it to the vendor or manufacturer and request an order response file. The manufacturer returns an order response file that lists the set of devices that are created. You then process this response file to store the data in the BRM database.

A policy opcode, PCM_OP_ORDER_POL_PROCESS, enables you to customize the **/order** object when you process an order.

Processing order response files includes creating devices in the database and updating the status of the requested order to either *Received* or *Partially Received*.

Managing the Order Life Cycle

Real-world orders pass through various stages. For example, an order can be new, received, or canceled. An order management system must keep track of these stages, or *order states*. In BRM, the order state is represented by the PIN_FLD_STATUS field in the **/order** object. The value of this field represents the current state of the order. The set of possible order states and state transitions defines the *life cycle* of an **/order** object.

Order life cycles are regulated by **/config/order_state** objects in the database. You define order states and state changes in a configuration file and load it into the database by using the **load_pin_order_state** utility. See "About Defining the Order Life Cycle" for more information.

Order management applications call PCM_OP_ORDER_UPDATE. This opcode is a wrapper opcode that calls one of the following opcodes to change order states, depending on the input flist provided for this opcode:

- PCM_OP_ORDER_SET_STATE
- PCM_OP_ORDER_SET_ATTR

PCM_OP_ORDER_UPDATE also validates the change specified in the opcode against the **/config/order_state** object for that order type. Invalid order states or state changes cause the opcode to fail.

Associating or Disassociating Orders with Master Orders

You can associate or disassociate orders with other orders. You can create orders and make them master orders by including or associating sub-orders with them. You can do this while creating the orders, or you can create orders and later associate them with other orders.

Applications call PCM_OP_ORDER_ASSOCIATE and the PCM_OP_ORDER_POL_ASSOCIATE policy opcode to associate and disassociate orders and create orders in different levels. A flag in the input flist determines whether the order is associated or disassociated. PCM_OP_ORDER_ASSOCIATE validates the associations against the order types, master orders, and sub-orders in question. Invalid associations cause the opcode to fail. PCM_OP_ORDER_ASSOCIATE updates the order array to include or exclude sub-orders. See "Associating and Disassociating /order Objects".

Managing Order Attributes

Each order has a unique ID, the *order ID*, that identifies the order for tracking by the order management system. This order ID is used to identify orders, search for orders, create a request file, modify orders, and delete orders.

> **Note:**
>
> To modify or delete an order, the state of the order must be *New*.

You can add fields to **/order** objects for attributes specific to particular order types.

Applications call PCM_OP_ORDER_SET_ATTR and the PCM_OP_ORDER_POL_SET_STATE policy opcode to set order attributes and order state. PCM_OP_ORDER_SET_ATTR can be used to change the common attributes mentioned above and the attributes introduced in subclasses. It cannot be used to change the order state or to cancel an order.

Deleting Orders

Orders can be deleted for a variety of reasons. You can call PCM_OP_ORDER_DELETE to remove the order object from the database when it is no longer needed.

During order deletion, the PCM_OP_ORDER_POL_DELETE policy opcode searches for the order by using the order ID. If the order ID is not found, the opcode generates an error. You can modify the policy to bypass the check and automatically remove the order object to be deleted.

> **Note:**
>
> If you delete a master order, the sub-orders associated with it are automatically deleted as well.

See "Associating or Disassociating Orders with Master Orders" for more information about master order–to–sub-order associations.

Tracking Order History

All order operations, such as order creation, state transitions, and changes to attributes, are recorded in the database as events. Recording events enables you to track the history of an order.

You can turn off event recording and turn on auditing to maintain order history without storing details. For information about choosing which events to record, see "Managing Database Usage" in *BRM System Administrator's Guide*. For an introduction to auditing and links to more detailed information, see "About Maintaining an Audit Trail of BRM Activity" in *BRM Managing Customers*.

Two BRM reports enable you to monitor the status and history of orders. See "About BRM Reports" in *BRM Reports*.

Managing the Order History Log

All operations related to the **/order** object, including order creation, setting order states, setting order attributes, and deleting orders, are logged in the database.

Customers can search the logging history for any **/order** object in the database.

Installing Order Manager

Order Manager is included in the Inventory Manager installation package.

To install Order Manager:

1. Follow the instructions in "Installing BRM" in *BRM Installation Guide* to install the full build package (**brmserver_15.0.0.*x*.0_linux_generic_full.jar**).
2. When the Installation Type screen appears during the installation process, do one of the following:
   - To install all BRM components, including the EAI Manager, select the **Complete** installation option.
   - To install the EAI Manager, along with other individual BRM components, select the **Custom** installation option, select **EAI Manager 15.0.0.*x*.0**, and then select any other optional managers you want to install.
3. Follow the instructions displayed during installation.

   > **Note:**
   >
   > The installation program does not prompt you for the installation directory if BRM or Inventory Manager is already installed on the machine and automatically installs the package at the *BRM_home* location, where *BRM_home* is the directory in which the BRM server software is installed.

4. Go to the directory where you installed the Inventory Manager package and source the **source.me** file:

   Bash shell:

   ```
   source source.me.sh
   ```

C shell:

```
source source.me.csh
```

5. Go to the *BRM_home*/**setup** directory and run the **pin_setup** script.

> **Note:**
>
> The **pin_setup** script starts all required BRM processes.

Uninstalling Order Manager

To uninstall Order Manager, run the *BRM_home*/**uninstaller/InventoryMgr/uninstaller.bin**.

Order Management and Multischema Environments

Order Manager supports multischema environments, with some restrictions:

- An **/order** object and the corresponding **/event/order** objects are stored in the specified database schema when an order is created.
- The master order object cannot be linked to sub-orders that reside in other schemas.
- You cannot move **/order** objects from one schema to another. You can delete the objects and re-create them in another schema, but master order and sub-order associations are lost.

About Defining the Order Life Cycle

You define the life cycle for order objects in BRM by editing or creating a configuration file and then loading the data into a **/config/order_state** object in the database.

Each **/config/order_state** object contains definitions of the possible order states and state changes for one order type. **/config/order_state** objects also contain information about which policy opcodes to call during state changes.

For detailed information about the syntax of the configuration file and running the load utility, see "load_pin_order_state".

> **Note:**
>
> You must load the life cycle definitions into the database *before* using any order management features. Because order management configuration data is always customized, no default values are loaded during BRM installation.

While the life cycle for every **/order** object is unique, the general pattern for all **/order** objects is the same:

- During order creation, an **/order** object moves from state *0* to an initial state that is defined for that order type.
- During the life of the **/order** object, it moves from state to state in the manner defined in a **/config/order_state** object. For example, an order might be able to move from the *New*

state to the *Request* state, but not directly from the *New* state to the *Received* state. Each order type has its own **/config/order_state** object, so you can define different life cycles to meet your business needs.

- At the end of the working life of the **/order** object, it moves into a final state such as *Received* or *Cancelled*. Depending on your business needs, the object could also be deleted to save space.

> **Note:**
>
> You can delete an order only when its order state is *New*.

There can be any number of order states, each of which is assigned a number in the order state configuration file. State 0 is reserved for the Raw state, but other numbers can be freely assigned.

There are four *state types* that correspond to the stages in the general pattern described above. These state types are defined in the **pin_order.h** header file and cannot be changed.

- State **0** indicates that the order is in Raw state.
- State **1** indicates that the order is in Init state.
- State **2** indicates that the order is in Normal state.
- State **3** indicates that the order is in End state.

When you define order states, they must be assigned to one of these types:

- **Raw**: There can be only one order state of this type for each order type. **/order** objects are in raw state only during the creation process. They can never be saved in the raw state. Raw states can transition only to Init states.
- **Init**: This state type is for states in which the object is first saved to the database. There can be more than one initial state for a particular order type. For example, you might want the order to be initialized to one state when created by a batch order creation tool and to another state when created by a CSR using a GUI tool. Init states can transition to *Init*, *Normal*, or *End* states. The order state *New* uses the *Init* state type.
- **Normal**: Normal order states are those that occur between the object's initial state and its end state. You can think of these as the working states for the order. There can be any number of Normal order states. *Normal* states can transition to *Init* states, other *Normal* states, and *End* states. The order states *Request* and *Partially Received* use the *Normal* state type.
- **End**: Orders cannot transition from *End* states. Because there are many possible end-of-life scenarios, you can include any number of *End* states. The order states *Received* and *Cancelled* use the *End* state type.

A typical order moves through the following states:

- **New**: The **/order** objects have this state when the order is created.
- **Request**: The **/order** objects have this state when an order is sent to the manufacturer for order creation.
- **Partially Received**: The **/order** objects have this state when the order is partially received.
- **Received**: The **/order** objects have this state when the order is completely received.
- **Cancelled**: The **/order** objects have this state when the order is canceled.

> **Note:**
>
> Only orders that have the order state *New* can be canceled.

When you define an order life cycle, be sure to consider all the stages in the life cycle of the order and all the possible relationships between them. You should also consider the level of detail that is useful to track in your order management system.

The names you use for order states can be localized for display in GUI applications. You must define the state names as text strings and load the definitions into the database by using the **load_localized_strings** utility.

Creating Custom Order Management Systems

You can create custom order management systems by using Order Manager. An order management system can be a separate, standalone application that performs all necessary tasks, or it can be an enabling application that connects BRM to an existing order management system.

"Adding New Client Applications" provides a general overview and information about writing applications in C. Other sections provide information about additional programming languages.

You need to complete some or all of the following tasks to implement an order management system:

1. Create a subclass of the **/order** storable class for every new order type. Each subclass should include fields for the unique attributes of the order type.

   You use Storable Class Editor to create fields and storable classes. See "Creating Custom Fields and Storable Classes".

2. If necessary, write a new Facility Module (FM) for any unique functionality required by your order management system.

   See "Writing a Custom Facilities Module".

3. If necessary, modify the Order FM policy opcodes to customize the default order management functionality.

   See "About System and Policy Opcodes" for general information and "Order FM Policy Opcodes" in *BRM Opcode Guide* for specific information about the Order Manager policy opcodes.

4. Write a custom application that calls the Order Manager standard and policy opcodes and any custom opcodes you created.

   See "Adding New Client Applications" for general information and information about writing applications in C.

5. Define the life cycles and order associations for your orders.

   See "About Defining the Order Life Cycle" and "Associating or Disassociating Orders with Master Orders".

About the Order Management Opcodes

Use the following opcodes to manage orders:

- To create an order, use PCM_OP_ORDER_CREATE. See "Creating /order Objects".

- To process an order request from a vendor or manufacturer, use PCM_OP_ORDER_PROCESS. See "Processing Order Response Files".
- To associate an order with a master order or a sub-order, use PCM_OP_ORDER_ASSOCIATE. See "Associating and Disassociating /order Objects".
- To update the state or attributes of an existing order, use PCM_OP_ORDER_UPDATE. See "Updating /order Objects".
- To delete an existing order, use PCM_OP_ORDER_DELETE. See "Deleting /order Objects".

> **Note:**
>
> These standard opcodes call Order FM policy opcodes prior to committing changes to the database. You can customize the Order FM policy opcodes to perform additional validation. See *BRM Opcode Guide*.

Creating /order Objects

Use PCM_OP_ORDER_CREATE to create an **/order** object. See "Creating Orders".

This opcode takes a type-only POID as input, which specifies the order type and the target database. PCM_OP_ORDER_CREATE performs these operations:

1. Calls the PCM_OP_ORDER_POL_CREATE policy opcode for validation. This opcode checks the POID type and calls other related Facilities Modules (FMs) and policy FMs to perform any validation checks those opcodes require.
2. Creates an **/order** object of the type specified in the input flist. This object includes all mandatory attributes for the order type.
3. Calls PCM_OP_ORDER_SET_STATE to set the initial order state. See "Setting the State in /order Objects".
4. If events are being recorded, generates an **/event/order/create** object.
5. If necessary, calls PCM_OP_ORDER_ASSOCIATE to associate the order with any master orders or sub-orders. See "Associating and Disassociating /order Objects".
6. Returns the POID of the **/order** object.

PCM_OP_ORDER_CREATE stops processing under these circumstances:

- When the PCM_OP_ORDER_POL_CREATE policy opcode fails.
- When PCM_OP_ORDER_SET_STATE fails.

If PCM_OP_ORDER_CREATE is not successful, it logs an error in the CM **pinlog** file, indicating the reason for the failure. The transaction is rolled back and no **/order** object is created.

Customizing Order Creation

Use the PCM_OP_ORDER_POL_CREATE policy opcode to customize any validation or checks to be performed before the actual processing starts.

For example, if orders of a particular type require an order ID with certain characteristics, you can validate the ID supplied by the input flist. Similarly, you can use the opcode to ensure that all mandatory attributes of a particular order type are included in the new object.

Processing Order Response Files

Use PCM_OP_ORDER_PROCESS to process order response files from the vendor or manufacturer. This opcode creates the devices in the BRM database and then updates the order status. See "Processing Order Response Files".

PCM_OP_ORDER_PROCESS performs these operations:

1. Retrieves data about the order from the **/order** object.
2. Calls the PCM_OP_ORDER_POL_PROCESS policy opcode for validation. This policy opcode calls other FMs and Policy FMs depending on the POID type passed in.
3. Calls PCM_OP_DEVICE_CREATE to create the specified devices in the BRM database. See "Managing Devices with BRM".
4. Calls PCM_OP_ORDER_UPDATE to update the status of the **/order** object.
5. If events are being processed, generates an **/event/order/process** object.
6. Returns the POID of the **/order** object.

PCM_OP_ORDER_PROCESS stops processing under these circumstances:

- When a **/config/order_state** object cannot be found for this order type.
- When the order state change is invalid based on the **/config/order_state** object for this order type.

Customizing Order Processing

The PCM_OP_ORDER_POL_PROCESS policy opcode allows you to customize any validation or checks to be performed before the actual processing starts.

You can use this opcode to perform validation of **/order** objects when you process the order.

Associating and Disassociating /order Objects

Use PCM_OP_ORDER_ASSOCIATE to associate or disassociate a sub-order with a master order. For information, see "Associating or Disassociating Orders with Master Orders".

You can associate or disassociate orders during account creation or when required. When associating sub-orders with master orders, it contains the POID of the master order.

PCM_OP_ORDER_ASSOCIATE performs these operations:

1. Determines whether to associate or disassociate by checking the PIN_FLD_FLAGS field:
   - When the flag is set to **1**, the opcode *associates* the specified objects.
   - When the flag is set to **0**, the opcode *disassociates* the specified objects.
2. For association only, validates the order–to–sub-order association.
3. Calls the PCM_OP_ORDER_POL_ASSOCIATE policy opcode for validation. This policy opcode calls other FMs and policy FMs depending on the POID type passed in.
4. Associates or disassociates the objects.
5. If events are being recorded, generates an **/event/order/associate** or **/event/order/disassociate** object.
6. Returns the POID of the **/order** object.

PCM_OP_ORDER_ASSOCIATE stops processing if an order listed in the input flist already exists in the **/order** object.

Customize How to Validate Association and Disassociation

Use the PCM_OP_ORDER_POL_ASSOCIATE policy opcode to customize any validation, in addition to the standard validation, or any checks to be performed before the actual order association or disassociation starts.

For example, you could limit the number of associations for particular order types or trigger a state change after certain associations or disassociations.

Updating /order Objects

Use PCM_OP_ORDER_UPDATE to update the state or attributes of an existing **/order** object.

This opcode takes as input the POID of the **/order** object to change, the program that is calling the opcode, and the fields to change, along with their new values. If the opcode is changing the attributes of an **/order** subclass, it can include fields introduced in the subclass.

PCM_OP_ORDER_UPDATE is a wrapper opcode that calls other opcodes to perform the actual modification to the order. The opcode determines whether to update the order state or order attributes by checking the fields passed in on the input flist:

- When the PIN_FLD_STATUS field is passed in, PCM_OP_ORDER_UPDATE changes the *order state* by calling PCM_OP_ORDER_SET_STATE. See "Setting the State in /order Objects".
- When the PIN_FLD_EXTENDED_INFO array is passed in, the opcode changes the *order attributes* by calling PCM_OP_ORDER_SET_ATTR. See "Changing /order Object Attributes".

PCM_OP_ORDER_UPDATE stops processing under these circumstances:

- When the PCM_OP_ORDER_POL_SET_ATTR or PCM_OP_ORDER_POL_SET_STATE policy opcode fails.
- When an attribute to change doesn't exist in the **/order** object.

Setting the State in /order Objects

Use PCM_OP_ORDER_SET_STATE to set or change the state of an **/order** object. During the processing of this opcode, there are two opportunities to call policy opcodes for validation:

- The first policy call takes place during the state change itself, just before the transaction is committed to the database.
- The second occurs just after the transaction that changes the order state.

You specify which policy opcodes to call for each state change in the same configuration file where you define the order life cycle. For each state change, you can specify an opcode for one, both, or neither of the two potential policy calls.

The PCM_OP_ORDER_POL_SET_STATE policy opcode is supplied as the default policy opcode for order state changes. You can create any number of custom policy opcodes to replace or supplement it. You enter the numbers of the policy opcodes in the configuration file.

For more information, see "Managing the Order Life Cycle".

PCM_OP_ORDER_SET_STATE takes as input the POID of the **/order** object, the calling program, the old state ID, and the new state ID. It then performs these operations:

1. Checks the validity of the state change based on the **/config/order_state** object for this order type. If the state change is not allowed, this opcode stops processing.
2. Calls the policy opcode specified in **/config/order_state** for the in-transition event.
3. Changes the PIN_FLD_STATUS field in the **/order** object to the new state.
4. Calls the policy opcode specified in **/config/order_state** for the post-transition event.
5. If events are being recorded, generates an **/event/order/state** object.
6. Returns the POID of the **/order** object.

PCM_OP_ORDER_SET_STATE stops processing under these circumstances:

- When the order state change is not valid based on the **/config/order_state** object for this order type.
- When a **/config/order_state** object cannot be found for this order type.
- When the input flist attempts to change the order from a *Raw* state type to any state type other than *Init* or attempts to change the order from an *End* state type to any other state.

Customizing How to Validate State Changes

The PCM_OP_ORDER_POL_SET_STATE policy opcode allows you to customize any validation or checks to be performed before the actual processing starts.

You can customize this policy opcode to provide additional validation or functionality during order state changes.

Changing /order Object Attributes

Use PCM_OP_ORDER_SET_ATTR to change the attributes for an existing **/order** object. This opcode is called by PCM_OP_ORDER_UPDATE to update the object's attributes. See "Managing Order Attributes".

Different order types can have different attributes. The attributes common to all orders, such as the text description stored in the PIN_FLD_DESCR field, are contained in the main **/order** object. Attributes specific to particular order types are stored in their subclasses.

> **Note:**
>
> You cannot use PCM_OP_ORDER_SET_ATTR to change the order state or order association. If the input flist includes these fields, they are ignored.

PCM_OP_ORDER_SET_ATTR performs these operations:

1. Calls the PCM_OP_ORDER_POL_SET_ATTR policy opcode for validation. This policy opcode calls other FMs and policy FMs based on the POID type passed in.
2. Updates the **/order** object to include the new attribute values.
3. If events are being recorded, generates an **/event/order/attribute** object.
4. Returns the POID of the **/order** object.

PCM_OP_ORDER_SET_ATTR stops processing under these circumstances:

- When the PCM_OP_ORDER_POL_SET_ATTR policy opcode fails.
- When an attribute to change doesn't exist in the **/order** object.

Customizing the opcode

The PCM_OP_ORDER_POL_SET_ATTR policy opcode allows you to customize any validation or checks to be performed before the actual processing starts.

You can use this opcode to perform validations or to set the attributes for the order object. For example, you can write code to validate the order ID in the input flist to conform to the pattern for a particular order type.

Deleting /order Objects

Use PCM_OP_ORDER_DELETE to delete an **/order** object. See "Deleting Orders".

PCM_OP_ORDER_DELETE performs these operations:

1. Calls the PCM_OP_ORDER_POL_DELETE policy opcode for validation. This policy opcode calls other FMs and policy FMs based on the POID type.
2. If the specified object is a sub-order, calls PCM_OP_ORDER_ASSOCIATE to disassociate the object from the master order.
3. Deletes the specified **/order** object from the BRM database.
4. If events are being recorded, generates an **/event/order/delete** object.
5. Returns the POID of the **/order** object.

PCM_OP_ORDER_DELETE stops processing if the PCM_OP_ORDER_POL_DELETE policy opcode fails.

Customizing How to Delete Orders

The PCM_OP_ORDER_POL_DELETE policy opcode allows you to customize any validation or checks to be performed before the actual processing starts.

You can use this policy opcode to customize the order-deletion process. For example, you can disable the service association check that is performed by default.

Part III

Integrating BRM with Enterprise Applications

This part describes the Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager framework and explains how to build a connector application to integrate BRM with other enterprise applications.

It contains the following chapters:

- About Enterprise Application Integration (EAI) Manager
- Installing EAI Manager
- Payload Configuration File Syntax
- Filtering which Business Events Are Published
- Building a Connector Application
- Configuring EAI Manager
- Configuring Business Events
- EAI DM Functions

27

About Enterprise Application Integration (EAI) Manager

Learn how to integrate Oracle Communications Billing and Revenue Management (BRM) with other enterprise applications by using the Enterprise Application Integration (EAI) Manager framework.

Topics in this document:

- About Integrating BRM with Enterprise Applications

About Integrating BRM with Enterprise Applications

You can integrate BRM with other applications in your enterprise by using EAI Manager. Integrating BRM with enterprise applications ensures data synchronization across applications in your enterprise and avoids data duplication among applications.

EAI Manager integrates BRM with enterprise applications by publishing business events. EAI Manager includes a default set of business events to be published. You can add additional events or remove unnecessary events by modifying a configuration file. A connector application built by you or a middleware vendor provides access to these events. The EAI Manager framework includes a set of functions that you implement in the connector application.

By default, EAI Manager publishes events in XML format. You can publish events in BRM flist format by specifying that format in the configuration file.

To collect and publish BRM events, EAI Manager performs these tasks:

1. EAI Manager uses event notification to cache events in the Payload Generator.
2. When the cached events form a complete business event as defined in the **payloadconfig.xml** configuration file, the Payload Generator generates the data (payload) to be published.

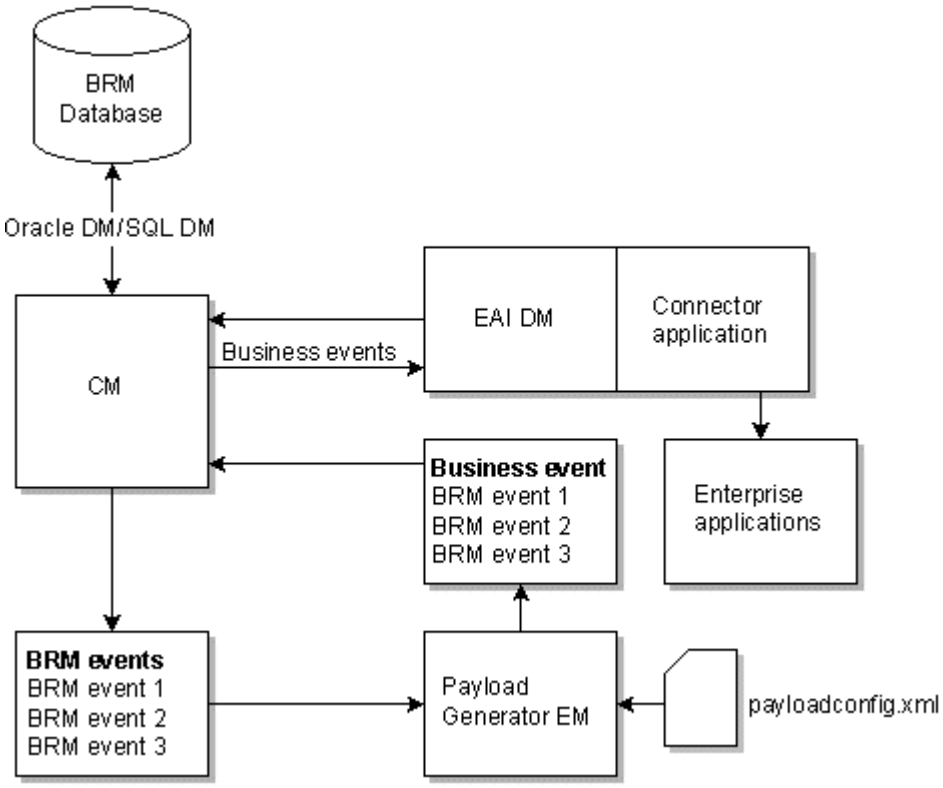
   The payload is generated in one of two ways, depending on the option you choose in the configuration file:

   - By reading the fields in the incoming flists for the event
   - By accessing the BRM database

   The Payload Generator generates the payload in XML or flist format as specified in the configuration file and sends it to the EAI Data Managers (DMs) through the Connection Manager (CM).
3. The EAI DM publishes the payload as a business event to external systems.

Figure 27-1 shows the EAI Manager architecture and data flow:

Figure 27-1 EAI Manager Architecture and Data Flow



To integrate BRM with enterprise applications, you need to perform the following tasks:

1. Install EAI Manager. See "Installing EAI Manager".
2. (Optional) Edit the configuration file to publish additional business events or change the data for default events. See "Configuring Business Events".
3. Build your connector application. See "Building a Connector Application".
4. Configure BRM to connect to EAI Manager and your connector application. See "Configuring EAI Manager".

28

Installing EAI Manager

Learn how to install the Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- About Installing EAI Manager
- Software Requirements
- Installing EAI Manager
- Configuring Event Notification for EAI Manager

About Installing EAI Manager

You can install EAI Manager on Linux operating systems. The EAI Manager includes these three EAI features:

- EAI Connection Manager (CM) module
- EAI Data Manager
- Payload Generator External Module (EM) (also called the EAI Java Server or **eai_js**)

When you install EAI Manager, the installation program assigns default values in the CM and EAI DM configuration (**pin.conf**) files and the Payload Generator properties (**Infranet.properties**) file. For information about changing the entries in the **pin.conf** and properties files, see "Configuring EAI Manager".

Software Requirements

Before installing EAI Manager, you must install:

- Third-Party software, which includes the Perl libraries and JRE required for installing BRM components. See "Preparing for BRM Installation" in *BRM Installation Guide*.
- BRM. See "BRM Installation Overview" in *BRM Installation Guide*.

Installing EAI Manager

To install EAI Manager:

1. Go to the directory where you installed the Third-Party package and source the **source.me** file.

> **Caution:**
>
> You must source the **source.me** file to proceed with installation, otherwise "suitable JVM not found" and other error messages appear.

Bash shell:

```
source source.me.sh
```

C shell:

```
source source.me.csh
```

2. Follow the instructions in "Installing BRM" in *BRM Installation Guide* to install the full build package (**brmserver_15.0.0.*x*.0_linux_generic_full.jar**).
3. When the **Installation Type** screen appears during the installation process, do one of the following:
   - To install all BRM components, including the EAI Manager, select the **Complete** installation option.
   - To install the EAI Manager, along with other individual BRM components, select the **Custom** installation option, select **EAI Manager 15.0.0.*x*.0**, and then select any other optional managers you want to install.
4. Follow the instructions displayed during installation.

> **Note:**
>
> The installation program does not prompt you for the installation directory if BRM or EAI Manager is already installed on the machine and automatically installs the package at the *BRM_home* location, where *BRM_home* is the directory in which the BRM server software is installed.

5. Go to the directory where you installed the EAI Manager package and source the **source.me** file:

   Bash shell:

   ```
   source source.me.sh
   ```

   C shell:

   ```
   source source.me.csh
   ```

6. Go to the *BRM_home*/**setup** directory and run the **pin_setup** script.
7. Open the *BRM_home*/**sys/cm/pin.conf** file in a text editor, change the value of the **enable_publish** entry to **1**, and then save and close the file.
8. Load information for EAI Manager into your system's event notification list.

   See "Configuring Event Notification for EAI Manager".
9. Stop and restart all BRM processes.

Increasing Heap Size to Avoid "Out of Memory" Error Messages

To avoid "Out of Memory" error messages, increase the maximum heap size used by the Java Virtual Machine (JVM). The exact amount varies greatly with your needs and system resources. By default, the JVM used has a maximum heap size of 60 MB. Increase the maximum heap size to 120 MB by entering the following sample code in a text editor:

```
%IF_EXISTS%("INIT_JAVA_HEAP", "@INIT_JAVA_HEAP@20m") %IF_EXISTS%
("MAX_JAVA_HEAP", "@MAX_JAVA_HEAP@120m")
```

where **20m** and **120m** indicate the minimum and maximum heap sizes respectively.

Save the file as *Packagename***.ja** in the temporary directory to which you downloaded the installation software. *Packagename* indicates the name of the installation software.

Configuring Event Notification for EAI Manager

When a BRM event that is included in a business event occurs, EAI Manager uses event notification to call the opcode that caches the BRM event in the Payload Generator.

Before you can use EAI Manager, you must configure the event notification feature as follows:

1. If your system has multiple configuration files for event notification, merge them. See "Merging Event Notification Lists".
2. Ensure that the merged file includes the entire event notification list in the *BRM_home***/sys/data/config/pin_notify_eai** file.

   > **Note:**
   >
   > The BRM events listed in the **pin_notify_eai** file are not business events. A business event comprises one or more BRM events. You define business events in the **payloadconfig.xml** file. See "Configuring Business Events".

3. If you configured EAI Manager to publish to an HTTP port, ensure that the merged file also includes the entire event notification list in the *BRM_home***/sys/data/config/pin_notify_plugin_http** file.

   See "Configuring EAI Manager to Publish to an HTTP Port".

4. (Optional) If necessary, add, modify, or delete entries in your final event notification list. See "Editing the Event Notification List".

   > **Note:**
   >
   > If you changed the default set of BRM events to be published by adding business events or by modifying the existing business events, you must edit your final event notification list to include all the BRM events in the new or modified business events.

5. Load your final event notification list into the BRM database. See "Loading the Event Notification List".

For more information, see "Using Event Notification".

29

Payload Configuration File Syntax

Learn about the payload configuration file syntax for Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- About the Payload Configuration File Syntax
- Publisher Definitions
- Event Definitions
- Element Definitions
- Syntax of Elements and Attributes
- Event Flist, Event Definition, and XML Output Example

About the Payload Configuration File Syntax

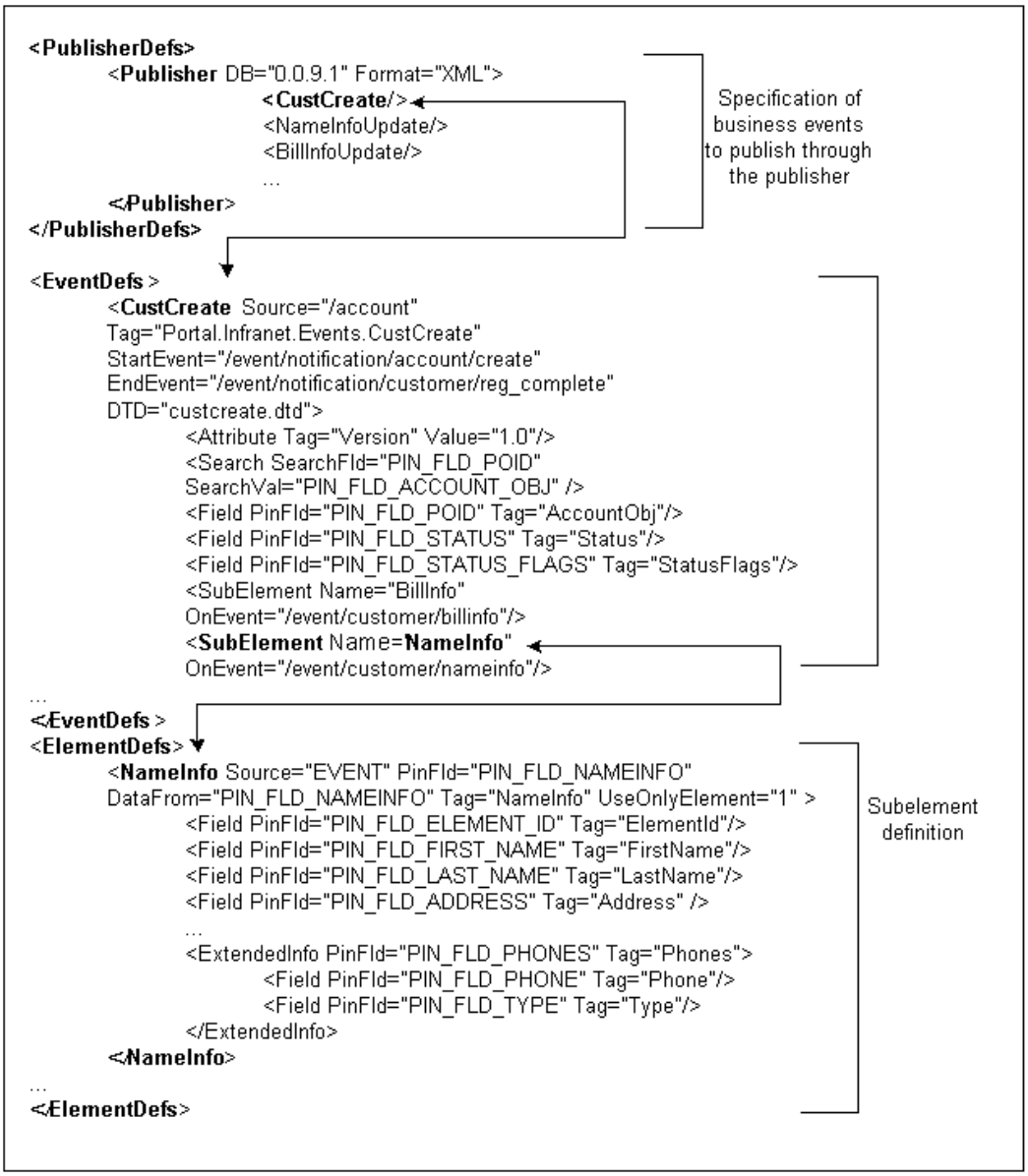
This section describes the syntax of the configuration file used to define the business events.

The configuration file has three sections:

- "Publisher Definitions" (**<PublisherDefs>**) lists the publisher and the events to publish. Each event in the list is defined in the event definition section.
- "Event Definitions " (<**EventDefs>**) defines the contents, source, and format of the events to publish. Each sub-element in the event definition is defined in the element definition section.
- "Element Definitions" (**<ElementDefs>**) defines the sub-elements specified in the event definitions.

Figure 29-1 shows the relationship between the elements in the different sections of the configuration file:

Figure 29-1 Sample Payload Configuration File



Publisher Definitions

You specify each business event that needs to be published, such as customer account creation, in the **<PublisherDefs>** section between the **<Publisher>** tags. The publisher definition is a list of all the business events to publish.

The **<PublisherDefs>** section defines the publisher to which the events are sent. You can include multiple publishers. You define each publisher as an element with the tag **<Publisher>**. You can also have multiple definitions for the same publisher, but with different output formats and tags.

The publisher definition includes the following elements:

- The mandatory attribute, **DB**, which specifies the database number of the EAI Data Manager (DM).

> **Note:**
>
> The database number must match the database number in the **dm_pointer** entry of the Connection Manager (CM) **pin.conf** file and the **dm_db_no** entry of the EAI DM **pin.conf** file.

- An optional attribute, **Format**, which specifies the format for the published output. Possible values are **XML**, which is the default, and **FLIST**.
- An optional attribute, **QTag**, which specifies the queue to which the business event will be enqueued. Oracle DM routes the business event to the queue with the same tag as the business event.
- A list of events to publish. Each event is listed in a separate tag. If an event triggers an identifier to be returned from a third-party application, the tag must include attributes specifying where the identifier will be stored in BRM. These attributes are required only for events that trigger identifiers.

  The following example includes business events that will be generated in multiple formats, **QTag** attributes that specify the queue to which the business events will be added, and one event (CustCreate) that triggers an identifier:

```
<PublisherDefs>
<Publisher DB="0.0.9.1" Format="XML" QTag="TAG1">
   <CustCreate ObjectPoid="PIN_FLD_POID"
IdentifierFld="PIN_FLD_ACCOUNT_NO"/>
   <NameInfoUpdate/>
   <BillInfoUpdate/>
...
</Publisher>

<Publisher DB="0.0.0.0" Format="Flist" QTag="TAG2">
   CustCreate,
   CustDelete,
   BillInfoUpdate,
   ProductPurchase,
   ProductCancel,
</Publisher>

</PublisherDefs>
```

> **Note:**
>
> In previous versions of EAI Manager, a different syntax was used for event definitions. The events were defined in a comma-separated list, not as tags. This syntax is still supported when *no* events trigger an identifier. If any of the defined events returns an identifier, all of the events must be defined with tags.

Event Definitions

Each event listed in the publisher definition is defined in the event definition between the **<EventDefs>** and **</EventDefs>** tags. The event definition contains a list of events to publish, each of which contains elements and attributes that specify the following information:

- Where to get the fields and the data for the event

- The sub-elements that make up the event
- The start and end event for the business event if multiple events are generated for the business event
- How the elements should be presented in the final XML output

> **Note:**
>
> If only one BRM event is generated during a business event, you need to specify only the start event.

The **<EventDefs>** tag defines a business event. Event definitions must have the following attributes:

- Source
- Tag
- StartEvent

Optionally, the event definition can include the following elements and attributes:

- EndEvent
- DTD
- Attribute
- PinFld
- Search
- SubElement
- ReadFlds
- ReadObj

This example shows the definition of the customer creation business event in the default configuration file:

```
<EventDefs>
   <CustCreate Source="/account"
   Tag="Portal.Infranet.Events.CustCreate"
   StartEvent="/event/notification/account/create"
   EndEvent="/event/notification/customer/reg_complete"
   DTD="custcreate.dtd">
   <Attribute Tag="Version" Value="1.0"/>
      <Search SearchFld="PIN_FLD_POID"
      SearchVal="PIN_FLD_ACCOUNT_OBJ"/>
      <Field PinFld="PIN_FLD_POID" Tag="AccountObj"/>
      <Field PinFld="PIN_FLD_STATUS" Tag="Status"/>
      <Field PinFld="PIN_FLD_STATUS_FLAGS" Tag="StatusFlags"/>
      <SubElement Name="BillInfo"
      OnEvent="/event/customer/billinfo"/>
      <SubElement Name="PayInfo"
      OnEvent="/event/notification/customer/reg_complete"/>
      <SubElement Name="NameInfo"
      OnEvent="/event/customer/nameinfo"/>
      <SubElement Name="DealInfo"
      OnEvent="/event/billing/deal/purchase"/>
      <SubElement Name="ProductInfo"
      OnEvent="/event/billing/product/action/purchase"/>
```

```
      <SubElement Name="ProductDetail"
      OnEvent="/event/billing/product/action/purchase"/>
      <SubElement Name="Service"
      OnEvent="/event/notification/customer/reg_complete"/>
      <SubElement Name="Profile"/>
   </CustCreate>
</EventDefs>
```

Element Definitions

Each sub-element in the event definition must be defined as an element in the element definition (**<ElementDefs>**). An array or substructure (at element level 0) in the BRM event flist or the object flist is defined in the element definition.

Each element definition specifies the following information:

- Where to get the data for the element
- The BRM field name for the data
- How to present the data in the final XML output

The **<ElementDefs>** tag is used to define elements. The element definition includes the following elements and attributes:

- The mandatory attributes, "Source" and "Tag"
- The optional attributes, "PinFld ", "UseOnlyElement", "PcmOpFlag", and "ReadValFrom"

> **Note:**
>
> If the output format for the payload is **FLIST**, **PinFld** is mandatory.

- The elements "Field", "ExtendedInfo", "ReadFlds", and "ReadObj"

This example shows the definition of two sub-elements, **NameInfo** and **BalInfo**, in the default configuration file:

```
<ElementDefs>
   <NameInfo Source="EVENT" PinFld="PIN_FLD_NAMEINFO"
   DataFrom="PIN_FLD_NAMEINFO" Tag="NameInfo" UseOnlyElement="1" >
      <Field PinFld="PIN_FLD_ELEMENT_ID" Tag="ElementId"/>
      <Field PinFld="PIN_FLD_FIRST_NAME" Tag="FirstName"/>
      <Field PinFld="PIN_FLD_MIDDLE_NAME" Tag="MiddleName"/>
      <Field PinFld="PIN_FLD_LAST_NAME" Tag="LastName"/>
      <Field PinFld="PIN_FLD_ADDRESS" Tag="Address"/>
      <Field PinFld="PIN_FLD_CITY" Tag="City"/>
      <Field PinFld="PIN_FLD_STATE" Tag="State"/>
      <Field PinFld="PIN_FLD_ZIP" Tag="Zip"/>
      <ExtendedInfo PinFld="PIN_FLD_PHONES" Tag="Phones">
          <Field PinFld="PIN_FLD_PHONE" Tag="Phone"/>
          <Field PinFld="PIN_FLD_TYPE" Tag="Type"/>
      </ExtendedInfo>
   </NameInfo>
   <BalInfo Source="/balance_group" PinFld="PIN_FLD_HEADER_INFO"
   PcmOpFlag="0x0002" Tag="BalInfo">
      <ReadFlds ObjPoid="PIN_FLD_BAL_GRP_OBJ"/>
      <Field PinFld="PIN_FLD_ACCOUNT_OBJ" Tag="AccObjs"/>
      <Field PinFld="PIN_FLD_NAME" Tag="Name"/>
      <Field PinFld="PIN_FLD_OBJECT_CACHE_TYPE" Tag="ObjCache"/>
```

```
      <ExtendedInfo PinFld="PIN_FLD_BALANCES" ElemId="840" Tag="Balance">
        <ExtendedInfo PinFld="PIN_FLD_SUB_BALANCES" Tag="SubBalance">
          <Field PinFld="PIN_FLD_CURRENT_BAL" Tag="CurrBal"/>
        </ExtendedInfo>
        <Field PinFld="PIN_FLD_CREDIT_PROFILE" Tag="CreditProfile"/>
      </ExtendedInfo>
      <Field PinFld="PIN_FLD_BILLINFO_OBJ" Tag="Billobj"/>
   </BalInfo>
</ElementDefs>
...
<ElementDefs>
 <BalInfo Source="/balance_group" PinFld="PIN_FLD_HEADER_INFO" Tag="BalInfo">
      <ReadObj ObjPoid="PIN_FLD_BAL_GRP_OBJ" ReadObjTag="BalGrpObj"/>
      <ExtendedInfo PinFld="PIN_FLD-BAL_INFO" Tag="BalInfo">
       <ReadObj ObjPoid="PIN_FLD_BILLINFO_OBJ" ReadObjTag="BillInfo"/>
      </ExtendedInfo>
 </BalInfo>
</ElementDefs>
```

Syntax of Elements and Attributes

This section describes the attributes and elements you use to define events and elements.

For an example of the relationship between the events, elements, and the flist data, see "Event Flist, Event Definition, and XML Output Example".

Source

Specifies where to get the data for fields specified in the event or element definitions.

Source can be a BRM storable class name or **EVENT**. If you set the **Source** attribute to **EVENT**, the Payload Generator EM reads the fields for the element from the incoming flist of the event. If you specify a storable class name, the Payload Generator EM reads the fields from the BRM database.

If all the data for the fields you need to publish is present in the event flist, specify **EVENT** for the source. If the event flist has a pointer to data through a POID (Portal object ID), specify **Source** to be a storable class.

For example:

```
<!-- Source is a Portal class name-->
PayInfo Source="/payinfo"

<!-- Source is the Event flist -->
NameInfo Source="EVENT"
```

Tag

Specifies the XML tag to be associated with the event, element, or field in the final XML output file.

For example, this entry in the configuration file appears as **<FirstName>Name</FirstName>** in the XML output:

```
<Field PinFld="PIN_FLD_FIRST_NAME" Tag="FirstName"/>
```

StartEvent

Specifies the BRM event that triggers the beginning of the business event. The Payload Generator EM starts caching event data when the event specified in the **StartEvent** attribute occurs. This element is mandatory in event definitions. If there is only one BRM event generated for the business event, the Payload Generator generates and publishes the business event as soon as it receives the event specified in **StartEvent**. If more than one BRM event is generated for a business event, the Payload Generator caches all the events starting with this event up to the event specified in **EndEvent**, and then generates and publishes the business event.

For example:

```
<-- StartEvent for customer creation business event -->
StartEvent="/event/notification/account/create"
```

EndEvent

Specifies the BRM event that marks the end of the business event. When the Payload Generator has cached all the events from the event specified in the **StartEvent** attribute to the event specified in the **EndEvent** attribute, the Payload Generator generates and sends the payload to the EAI DM.

When this attribute isn't present, the business event begins and ends with the BRM event specified in **StartEvent**.

For example:

```
<!-- EndEvent for customer creation business event -->
EndEvent="/event/notification/customer/reg_complete"
```

DataFrom

Specifies the array or substructure field in an flist from which to retrieve data for the event or the element.

If the source for the element data is specified as **EVENT**, the array or substructure is read from the incoming flist for the event. If the source for the element data is specified as a storable class, the array or substructure is read from the storable class in the BRM database.

For example:

```
<!-- Data to be read from PIN_FLD_NAMEINFO array for the NameInfo element -->

DataFrom="PIN_FLD_NAMEINFO"
```

UseOnlyElement

Specifies an element of an array to publish. Use this tag when you want to publish only one of the elements of the array in the flist.

For example, suppose the **BillInfo** element of a business event is constructed when an **/event/customer/billinfo** event occurs. To publish only element 1, specify **UseOnlyElement="1"**.

UseElementId

Determines whether element IDs are retained with array fields. If you set **UseElementId** to **1**, element IDs are retained with array elements. If you set **UseElementId** to **0**, array elements are published in sequential order without their original element IDs.

> **Note:**
>
> If the data is published in flist format with **UseElementID** set to **0**, elements are numbered sequentially starting from 0. These numbers do not correspond to the original element IDs. They are included only to ensure that the flist is in a valid format.

For example:

```
<!-- This tag will cause element IDs to be retained for elements in this array -->

<NameInfo Source="EVENT" PinFld="PIN_FLD_NAMEINFO"
DataFrom="PIN_FLD_NAMEINFO" Tag="NameInfo" UseElementId="1" >
```

Attribute

Adds an attribute to the XML output of the events.

> **Note:**
>
> **Attribute** is used only in event definitions.

You must specify two entries for the **Attribute** field:

- **Tag** to specify the name of the attribute to add.
- **Value** to specify the value of the attribute.

For example:

```
<!-- This Attribute adds the version number to a customer creation business event
definition -->
<CustCreate ... Tag="Portal.Infranet.Events.CustCreate"
<Attribute Tag="Version" Value="1.0"/>
```

The XML output for this business event includes a **Version** attribute:

```
<Portal.Infranet.Events.CustCreate Version="1.0">
```

DTD

Specifies the data type definition (DTD) used for the XML output for this business event.

> **Note:**
>
> You need to specify a DTD only if you publish valid XML documents.

You must use the **Name** attribute to specify the name of the DTD file.

This example specifies that the **custcreate.dtd** file be used for the output:

```
<!-- Specifies the custcreate.dtd as the file to use for the XML output -->

<DTD Name="custcreate.dtd">
```

The XML output for that example includes the DTD file in its DOCTYPE tag:

```
<!DOCTYPE Portal.Infranet.Events.CustCreate SYSTEM "custcreate.dtd">
```

PinFld

Specifies the BRM field name.

For element and **ExtendedInfo** definitions, the data type of the BRM field must be an array or substructure.

> **Note:**
>
> This field is mandatory if you specify **FLIST** as the output format for the published event.

```
<--! Specifies that the data for the NameInfo element is from the PIN_FLD_NAMEINFO field
of the event flist-->

<NameInfo Source="EVENT" PinFld="PIN_FLD_NAMEINFO"
```

Field

Specifies the BRM field to include in the definition of an event, element, or **ExtendedInfo**.

You must specify two attributes:

- **PinFld** to specify the name of the field in BRM.
- **Tag** to specify the XML tag to use for the value in the XML file generated for this business event.

This example specifies that the data is retrieved from the PIN_FLD_FIRST_NAME field and assigned to the **<FirstName>** tag in the XML output:

```
<Field PinFld="PIN_FLD_FIRST_NAME" Tag="FirstName"/>
```

The XML output:

```
<FirstName> Name </FirstName>
```

ExtendedInfo

Specifies an array or substruct field within an element definition.

For example, if the array that defines the element contains a nested array that you want to include in your element definition, use **ExtendedInfo**.

You must use the following attributes:

- **PinFld** to specify the BRM field name. The data type must be an array or substruct.
- **Tag** to specify the XML tag used for the value in the XML file generated for this business event.

You can optionally specify the following attributes:

- "UseOnlyElement" to read only the element specified from the array.
- "ElemId" to retrieve a specific array element. If you do not include **ElemId**, all elements within that array are returned.

Search

Specifies the search criteria to read data from a storable class when **Source** is specified as a storable class.

> **Note:**
>
> You can specify **Search** only in event and element definitions.

EAI Manager searches the database by creating an SQL query using the attributes specified:

- **SearchFld** specifies the field to search in the database.
- **SearchVal** specifies the value in the input flist to be used in the search arguments.
- **SearchValFrom** specifies that the value for the search needs to be read from an array or substruct element in the input flist.

For example, when this **Search** field is specified:

```
<PayInfo Source="/payinfo" PinFld="PIN_FLD_PAYINFO"
   Tag="PayInfo" >
      <Search SearchFld="PIN_FLD_ACCOUNT_OBJ"
      SearchVal="PIN_FLD_ACCOUNT_OBJ"/>
...
```

EAI Manager creates this query:

Select from **/payinfo** where PIN_FLD_ACCOUNT_OBJ equals the value of the field PIN_FLD_ACCOUNT_OBJ in the event flist.

When this **Search** field is specified:

```
<ProductDetail Source="/product" PinFld="PIN_FLD_PRODUCT"
   Tag="ProductDetail">
      <Search SearchFld="PIN_FLD_POID"
      SearchVal="PIN_FLD_PRODUCT_OBJ"
```

```
      SearchValFrom="PIN_FLD_PRODUCT"/>
....
```

EAI Manager creates this query:

Select from **/product** where PIN_FLD_POID equals the value of the field PIN_FLD_PRODUCT_OBJ in the PIN_FLD_PRODUCT array of the event flist.

SubElement

Specifies the element to add to an event in the event definition.

You must specify the **Name** attribute, which must match the tag of the element in the **<ElementDefs>** section. If you have an element called **NameInfo** in the **<ElementDefs>** tag:

```
<ElementDefs>
<NameInfo Source="EVENT"...>
```

the **Name** attribute in the **SubElement** must be:

```
<SubElement Name="NameInfo">
```

You can optionally include the **OnEvent** attribute if the **Source** for the sub-element is **"Event**". If you specify **OnEvent**, the data for the sub-element is generated when the event occurs.

If the **Source** for the data is a storable class, the data is read from the database before the business event is published.

For example, this entry specifies that the data for the **NameInfo Sub-element** is read when the **nameinfo** event occurs in BRM:

```
<SubElement Name="NameInfo"
OnEvent="/event/customer/nameinfo"/>
```

PcmOpFlag

Specifies an optional field that stores the flag value to run the PCM_OP_READ_FLDS or PCM_OP_READ_OBJ opcode. It is specific to read-only objects.

If you do not specify a value for **PcmOpFlag**, the default value is **CACHEABLE**.

> **Note:**
>
> **PcmOpFlag** is supported only when Source is set to a storable class name and not when it is set to **EVENT**.

For example:

```
PcmOpFlag="0x0002"
```

ReadFlds

Specifies to read one or more fields in an object from the transaction cache or, if it is not in the cache, from the BRM database.

You must use the **ObjPoid** attribute to specify the field containing the object POID to read.

For example:

```
<ReadFlds ObjPoid="PIN_FLD_BAL_GRP_OBJ"/>
```

ReadObj

Specifies to read an entire object from the transaction cache or, if it is not in the cache, from the BRM database.

You must specify two attributes:

- **ObjPoid** to specify the field containing the POID of the object to read.
- **ReadObjTag** to specify the XML tag to use for the value in the XML file generated for this business event.

For example:

```
<ReadObj ObjPoid="PIN_FLD_BAL_GRP_OBJ" ReadObjTag="balGrpObj"/>
```

> **Note:**
>
> You cannot perform ReadFlds, ReadObj, and Search at the same time. If you want to perform all three together, you need to pass each of them as a SubElement.

ElemId

Specifies to read a specific array element.

> **Note:**
>
> The **ElemId** attribute is used only in **ExtendedInfo** elements.

For example, this entry specifies to read the PIN_FLD_BALANCES array only when its element ID is 840. The data is assigned to the **<balance>** tag in the XML output:

```
<ExtendedInfo PinFld="PIN_FLD_BALANCES" ElemId="840" Tag="balance">
```

ReadValFrom

Specifies an optional attribute that retrieves an object POID (**ObjPoid**) from a specific array in the incoming opcode flist. If you do not specify the value of the **ReadValFrom** attribute, the object POID is retrieved from the 0th level.

> **Note:**
>
> The **ReadValFrom** attribute is used in **ReadFlds** and **ReadObj** elements.

This example specifies that the PIN_FLD_BAL_GRP_OBJ **ObjPoid** is retrieved from the PIN_FLD_BAL_IMPACTS array:

```
<ReadObj ObjPoid="PIN_FLD_BAL_GRP_OBJ" ReadObjTag="BalGrpObj"
ReadValFrom="PIN_FLD_BAL_IMPACTS"/>
```

Event Flist, Event Definition, and XML Output Example

The following figures show the **NameInfoUpdate** business event definition, how the data in the event flist is used to construct the business event, and the published XML document.

> **Note:**
>
> These examples show only a few of the fields in the event flist and event definition.

Figure 29-2 shows a partial NameInfoUpdate event flist and the NameInfoUpdate definition:

Figure 29-2 NameInfoUpdate Event Flist and Event Definition

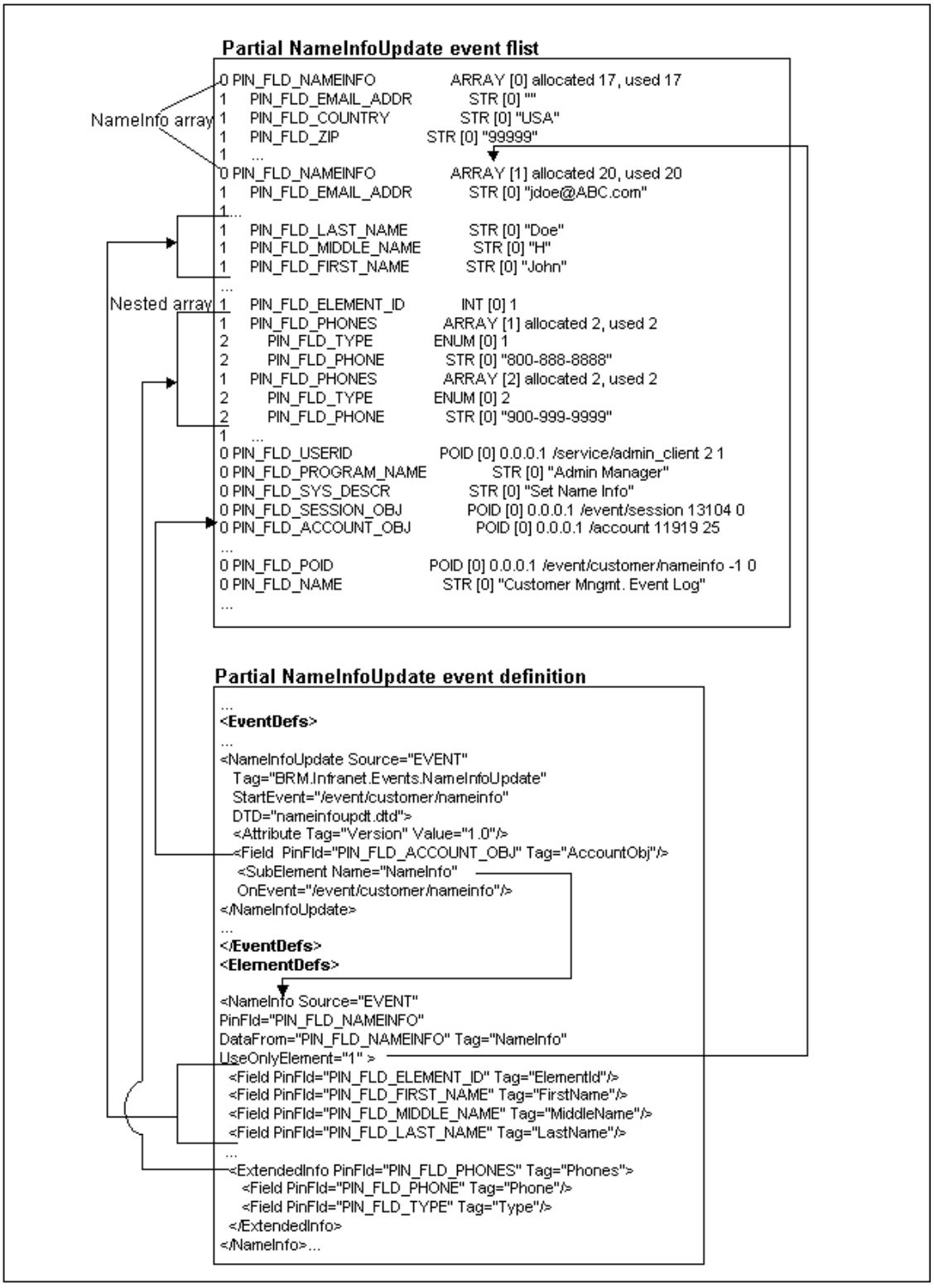


Figure 29-3 shows the published XML output of the **NameInfoUpdate** event in the previous example:

Figure 29-3 NameInfoUpdate XML Output

XML output of the NameInfoUpdate event

```
<?xml version="1.0"?>
<!DOCTYPE Portal.Infranet.Events.NameInfoUpdate
SYSTEM "nameinfoupdt.dtd" >
<Portal.Infranet.Events.NameInfoUpdate
Version="1.0">
<AccountObj>0.0.0.1 /account 11919 25</AccountObj>
  <NameInfo>
    <ElementId>1</ElementId>
    <FirstName>John</FirstName>
    <MiddleName>H</MiddleName>
    <LastName>Doe</LastName>
     <Phones>
        <Phone>800-888-8888</Phone>
        <Type>1</Type>
    </Phones>
     <Phones>
        <Phone>900-999-9999</Phone>
        <Type>2</Type>
     </Phones>
  </NameInfo>
</Portal.Infranet.Events.NameInfoUpdate>
```

30

Filtering which Business Events Are Published

Learn how to filter business events for Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- Filtering which Business Events Are Published
- About the Condition Attribute
- About the Condition Definition

Filtering which Business Events Are Published

You can configure EAI Manager to filter which business events are published or not published based on business event attributes. You define the criteria a business event must meet, and only business events meeting the criteria are published.

You filter which business events are published and not published by creating a condition, which consists of the following:

- The **Condition** attribute. See "About the Condition Attribute".
- The criteria that the business event must meet to be published. See "About the Condition Definition".

About the Condition Attribute

The **Condition** attribute specifies the name of your condition. You can use any name, such as MyCondition. You specify the condition name in the **<PublisherDefs>** section of the **payloadconfig.xml** file using the following format:

```
Condition=ConditionName
```

You indicate whether to apply the condition to a specific business event or all business events in the publisher through the placement of the **Condition** attribute:

- For a specific business event, you add the **Condition** attribute after the appropriate business event. For example, to apply the condition to **BillInfoUpdate** business events:

  ```
  <PublisherDefs>
     <Publisher DB="0.0.0.0" Format="FLIST">
        CustCreate,
        CustDelete,
        BillInfoUpdate Condition=MyCondition
     </Publisher>
  </PublisherDefs>
  ```

  In this example, only the **BillInfoUpdate** business event will be filtered by MyCondition.

- For all business events in the publisher, you add the **Condition** attribute, surrounded by quotes, to the **<Publisher>** tag. For example:

```
<PublisherDefs>
   <Publisher DB="0.0.0.0" Condition="MyCondition" Format="FLIST"/>
      CustCreate,
      CustDelete,
      BillInfoUpdate
   </Publisher>
</PublisherDefs>
```

In this example, the **CustCreate**, **CustDelete**, and **BillInfoUpdate** business events will be filtered by MyCondition.

About the Condition Definition

The condition definition defines the criteria that a business event must meet to be published. The condition definition must appear at the end of the **payloadconfig.xml** file in its own **<ConditionDefs>** section and follow this format:

```
<ConditionDefs>
   <ConditionName>
      <BooleanOperator>
         <BooleanExpression PinFld="FieldName" Value="FieldValue" Operator="OpValue"/>
      </BooleanOperator>
   </ConditionName>
</ConditionDefs>
```

Table 30-1 lists the elements in the **<ConditionDefs>** section.

Table 30-1 Elements in ConditionDefs

Element	Syntax	Description
ConditionName	`<MyCondition>`	Opens the condition definition. *ConditionName* must match the name you defined in the **Condition** attribute. See "About the Condition Attribute".
BooleanOperator	`<AndExpression>`	The Boolean operator to apply. Possible values are: • **AndExpression** is the Boolean AND. • **OrExpression** is the Boolean OR. • **NotExpression** is the Boolean NOT.

Table 30-1 (Cont.) Elements in ConditionDefs

Element	Syntax	Description
BooleanExpression	`<BooleanExpression PinFld="PIN_FLD_FLAGS" Value="128" Operator="GT"/>`	Contains the definition for the Boolean expression, including the field name, field value, and operator. **PinFld** specifies the business event flist field name. You can also use OP_FLAGS, which is the flag with which the business event was generated. **Value** specifies the value of the business event flist field. **Operator** can be one of the following: • **EQ** is equal to. • **NE** is not equal to. • **LT** is less than. • **GT** is greater than. • **LE** is less than. • **GE** is greater than or equal to. • **CHECK-BITS** checks the specified bits for Integers. • **CHECK-DB** checks the database specified in the POID field. • **CHECK-ID** checks the ID of the POID field. • **CHECK-TYPE** checks the type of the POID field. **Note:** The operators that can be used with each field depend on the field's data type: – INT and ENUM data types support EQ, NE, GT, LT, GE, LE, and CHECK-BITS operators. – POID data types support EQ, NE, CHECK-DB, CHECK-ID, and CHECK-TYPE operators. – STR data types support EQ and NE operators. – DECIMAL and TSTAMP data types support EQ, NE, GT, LT, GE, LE operators.

31
Building a Connector Application

Learn how to set up enterprise applications to access Oracle Communications Billing and Revenue Management (BRM) event data by using a connector application for Enterprise Application Integration (EAI) Manager.

Topics in this document:

- Building a Connector Application

Building a Connector Application

To provide access to BRM event data for enterprise applications, you need to build a module connector application that handles transaction management and transformation schemes specific to your environment.

The EAI DM calls the functions listed in Table 31-1 when it starts, processes event data, and shuts down. You must implement these functions in your connector application.

Table 31-1 Functions Called by EAI Data Manager

Function	Description
AbortTransaction	Called by the EAI DM after the transaction is cancelled in BRM.
CommitTransaction	Called by the EAI DM after BRM commits the transaction to its database.
FreeGlobalContext	Called when the EAI DM process is shutting down.
GetGlobalContext	Called from a connector application to access the context initialized with **InitializeGlobalContext**.
Initialize	Called by the EAI DM when it starts.
OpenTransaction	Called by the EAI DM before it calls **PublishEvent**.
PrepareCommit	Called by the EAI DM when BRM is about to commit the transaction to the database.
PublishEvent	Called by the EAI DM when there is a business event to be published.
SetIdentifier	Called from the connector application to set a return identifier for a published business event.
Shutdown	Called by the EAI DM when it shuts down.

The **plugin_flist.c** and **plugin_xml.c** files in the *BRM_home*/**sys/dm_eai** directory provide sample implementations of the EAI functions in flist and XML format.

After you build your connector application, configure EAI Manager. See "Configuring EAI Manager".

32
Configuring EAI Manager

Learn how to configure Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- Configuring the Connection Manager for EAI
- Configuring the EAI DM
- Configuring the Payload Generator EM
- Specifying the Date and Time Format for Business Events
- Defining Infinite Start Date and End Date Values
- Configuring EAI Manager to Publish to an HTTP Port

When you install EAI Manager, the installation program adds entries to the EAI DM and the CM **pin.conf** files and to the **Infranet.properties** file. After you build the connector application, you need to enable EAI by editing the **pin.conf** and **Infranet.properties** files to specify values for the entries relevant to EAI Manager.

Configuring the Connection Manager for EAI

After you build the connector application, you need to configure the CM for EAI.

1. In a text editor, open the CM configuration file (*BRM_home***sys\cm\pin.conf**).

   This **pin.conf** file contains descriptions of all the entries and instructions for editing the entries.

2. In the EAI_CM section of the file, assign values to the entries shown in Table 32-1:

Table 32-1 EAI CM Entries

Entry	Description	Example
- fm_publish enable_publish	Specifies whether to publish events. To publish events, specify **1**. To not publish events, specify **0**.	**- fm_publish enable_publish 1**

Table 32-1 (Cont.) EAI CM Entries

Entry	Description	Example
- cm em_pointer	Specifies the host name and port number of the computer on which the Payload Generator EM runs. The **em_pointer** entry includes the following values: • Tag that refers to the EM type; for example, **publish** for the EAI EM. • The address type tag, **ip**. • IP address or host name of the computer where the Payload Generator EM runs. • Port number of the computer where the Payload Generator EM runs. **Important**: The port number must match the port number in the **Infranet.properties** file in the *BRM_home***sys\eai_js** directory.	**- cm em_pointer publish ip 127.0.0.1 11930**
- cm em_group	Specifies a member opcode in a group of opcodes provided by the Payload Generator EM. **Note**: This entry is automatically inserted in the configuration file by the installation program. The **em_group** entry includes the following values: • Tag that refers to the EM type; for example, **publish** for the EAI EM. • The opcode name or number.	**- cm em_group publish 1301**
- cm dm_pointer	Specifies the database number, the IP address or host name, and port number of the computer where the EAI DM runs. **Important**: The database number must match the **DB** entry of the publisher definition in the **payloadconfig.xml** file. The database number and port number must match the **dm_db_no** and **dm_port** entries in the EAI DM **pin.conf** file.	**- cm dm_pointer 0.0.9.1 ip 127.0.0.1 11970**

3. Save and close the file.
4. Stop and restart the CM:

```
cd BRM_home/bin
pin_ctl bounce cm
```

Configuring the EAI DM

After you build the connector application, you need to configure the EAI DM.

1. In a text editor, open the EAI DM configuration file (*BRM_home***sys\dm_eai\pin.conf**).
2. In the EAI_PINCONF entries section, assign values to the entries shown in Table 32-2:

Table 32-2 EAI_PINCONF Entries

Entry	Description	Example
- dm plugin_name	Specifies the name of the module connector application that you implemented.	**- dm plugin_name ./dm_eai_plugin.so**
- dm dm_db_no	Specifies the database number assigned to the EAI DM. The format of the entry is **0.0.0.***n***. 0**, where *n* is your database number. **Important**: This number must match the **DB** entry for the publisher definition in the **payloadconfig.xml** file and the **dm_pointer** entry in the CM **pin.conf** file.	**- dm dm_db_no 0.0.9.1 0**
- dm dm_port	Specifies the port number of the computer where the EAI DM runs. **Important**: This number must match the port number in the **dm_pointer** entry in the CM **pin.conf** file.	**- dm dm_port 11970**
- dm loglevel	Specifies the log level of the EAI DM: **0** = no logging **1** = log only error messages (default) **2** = log error messages and warnings **3** = log error messages, warnings, and debugging messages	**- dm loglevel 1**

3. Save and close the file.
4. Stop and restart the EAI DM:

```
cd BRM_home/bin
pin_ctl bounce dm_eai
```

Configuring the Payload Generator EM

The Payload Generator has an **Infranet.properties** file that specifies the location of the **payloadconfig.xml** file:

To configure the Payload Generator:

1. In a text editor, open the Payload Generator configuration file (*BRM_home***sys\eai_js\Infranet.properties**).
2. Specify the name and location of the **payloadconfig.xml** file:
   - If you are *not* using the **plugin_http** module, verify that the **infranet.eai.configFile** entry points to the location of the **payloadconfig.xml** file.

     > **Note:**
     >
     > If you edited the **payloadconfig.xml** file and saved it with a different name, make sure you also change the name in this entry.

   - If you *are* using the **plugin_http** module, change the **infranet.eai.configFile** entry to point to the location of the **payloadconfig_plugin_http.xml** file.

3. Verify that the file contains the following entry:

   ```
   infranet.opcode.handler.PUBLISH_GEN_PAYLOAD=com.portal.eai.PublishHandler
   ```

4. Verify that the port number specified in the **infranet.server.portNR** entry matches the port number in the **em_pointer publish** entry in the CM **pin.conf** file.
5. Save and close the file.
6. Stop and restart the Payload Generator EM:

   ```
   cd BRM_home/bin
   pin_ctl bounce ePai_js
   ```

Specifying the Date and Time Format for Business Events

In business events, the date field value uses the default EAI Manager format in the server's local time zone. You can configure the date field to use a different date and time format by using the following entry in the Payload Generator **Infranet.properties** file:

infranet.eai.date_pattern: Specifies the date and time format based on the ISO-8601 standard. For example, you can set this entry to any of the following formats:

- **infranet.eai.date_pattern=dd/MMM/yyyy:hh:mm:ss**
- **infranet.eai.date_pattern=yyyy-MM-dd'T'hh:mm:ss**. Use this format if EAI Manager uses Oracle AIA to exchange data with external applications.

To specify the date and time format in business events:

1. In a text editor, open the Payload Generator configuration file (*BRM_home***sys\eai_js\Infranet.properties**).
2. Specify the date format in the **infranet.eai.date_pattern** entry:

   ```
   infranet.eai.date_pattern = Format
   ```

3. Save and close the file.
4. Stop and restart the Payload Generator EM:

   ```
   cd BRM_home/bin
   pin_ctl bounce eai_js
   ```

Defining Infinite Start Date and End Date Values

In some external applications, the infinite date value is represented as a NULL (empty XML element) value and in other external applications as the epoch time (01-01-1970 1200 AM UTC).

By default, when EAI Manager sends data to your external application, the infinite date value is the start of the epoch time.

You can define how EAI Manager sets infinite date values by using the **infranet.eai.xml_zero_epoch_as_null** entry in the Payload Configurator **Infranet.properties** file.

> **Note:**
>
> The **infranet.eai.xml_zero_epoch_as_null** entry does not affect the flist payload.

To configure how EAI Manager sets infinite date values:

1. Open the *BRM_home***/sys/eai_js/Infranet.properties** file in a text editor.
2. Add the following entry:

   ```
   infranet.eai.xml_zero_epoch_as_null = value
   ```

   where *value* is:

   - **TRUE** to use NULL to represent an infinite start or end date.
   - **FALSE** to use the epoch time to represent an infinite start or end date. This is the default.
3. Save and close the file.
4. Stop and restart the Payload Generator EM:

   ```
   cd BRM_home/bin
   pin_ctl bounce eai_js
   ```

Configuring EAI Manager to Publish to an HTTP Port

You can use EAI Manager to publish information from your BRM database to an HTTP port for use by a third-party application. For example, you can send charge offer information to a customer relationship manager (CRM), such as Siebel Communications. Information about the new charge offers is posted to a specific HTTP port to enable the CRM to create charge offer information.

To configure EAI Manager to publish to an HTTP port:

1. Configure Connection Manager for EAI. See "Configuring the Connection Manager for EAI".
2. Open the **dm_eai** configuration file (*BRM_home***\sys\dm_eai\pin.conf**) with a text editor such as **vi**.
3. Add the following line to specify the name of the **dm_http** module:

   ```
   - dm plugin_name plugin_http.extension
   ```

   where *extension* is the library extension for your operating system: **so** for Linux.
4. Add the following line to configure the header delimiter:

   ```
   - dm dm_http_delim_crlf value
   ```

   where *value* is:

   - **0** to specify the delimiter \n
   - **1** to specify the delimiter \r\n

   The default is **0**.
5. Specify the HTTP host name and port number of the server to which the data should be sent:

   ```
   - dm dm_http_agent_ip host_name port_number
   ```
6. If required, specify the URL for the HTTP server; for example:

   ```
   - dm dm_http_url http://10.1.6.78/HTTP_Infranet/BTSHTTPRECEIVE.so
   ```

> **Note:**
>
> The URL might be required; for example, when you deploy EAI Manager in an IIS environment.

7. If your HTTP server requires the host name in the header, add this line to the **pin.conf** file:

   ```
   - dm dm_http_header_send_host_name value
   ```

   where *value* is:

   - **0** to indicate that the host name won't be included in the header.
   - **1** to indicate that the host name will be included in the header.
   - **2** to indicate that both the host name and the port number will be included in the header.

   The default is **0**.

8. If your HTTP server sends a **100-Continue** status code to clients that do not send a 100-Continue expectation, add this line to the **pin.conf** file:

   ```
   - dm dm_http_100_continue value
   ```

   where *value* is:

   - **0** to indicate that a 100-Continue status is not expected.
   - **1** to indicate that a 100-Continue status is expected.

   > **Note:**
   >
   > Set *value* to **1** only if your server sends the 100-Continue status code to clients that do not send an expectation for it. Most servers do not send unexpected 100-Continue codes, but some do.

   The default is **0**.

9. Specify whether or not **dm_http** should read the response codes sent by the HTTP server:

   ```
   - dm dm_http_read_success value
   ```

   where *value* is:

   - **0** to indicate that the module should not wait for a response code from HTTP receiver.
   - **1** to indicate that the module should wait for a response code to be read from HTTP receiver.

   The **dm_http** module supports the success response codes **200** (OK) and **202** (request accepted for asynchronous processing).

   The default is **0**.

10. Save and close the file.
11. Configure the payload generator. See "Configuring the Payload Generator EM".
12. Load information for the **dm_http** module into your system's event notification list.

    See "Configuring Event Notification for EAI Manager".

13. Stop and restart BRM.

33
Configuring Business Events

Learn how to configure business events for Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- About BRM Business Events
- About Publishing Additional Business Events
- Setting Up Multiple Publishers and Events
- Defining Business Events
- Removing Events That You Do Not Want to Publish
- Returning Identifiers from Enterprise Applications
- Changing the Format of Published Events
- Validating Your Changes to the Payload Configuration File

About BRM Business Events

A business event is a BRM operation that you define in the Payload Generator EM configuration file (**payloadconfig.xml**). A number of business events are defined by default; for example, one of the default business events is ProductPurchase, which is created when a customer buys a charge offer.

A business event is created only after EAI Manager has been notified that a qualifying BRM event has occurred. For example, several BRM events, including **/event/customer/billinfo**, **/event/billing/product/action/purchase**, and **/event/customer/nameinfo**, must occur before the CustCreate business event is created.

BRM uses event notification to cache the events that make up a business event in the Payload Generator. See "Configuring Event Notification for EAI Manager".

Business event definitions include data or pointers to data from the flists of events included in the definition. For an example of the relationship between the event flist and the definition of the business event to publish, see "Event Flist, Event Definition, and XML Output Example".

The default set of business events that BRM publishes is defined in the Payload Generator EM configuration file, **payloadconfig.xml**. You can edit the configuration file to:

- Add events that you want to publish.
- Remove events that you do not want to publish.
- Specify whether you want the events to be published in XML or flist format.

For the definitions of the default set of business events, see the payload configuration file (*BRM_home***/sys/eai_js/payloadconfig.xml**).

About Publishing Additional Business Events

To publish additional business events, you include definitions of the events in the **payloadconfig.xml** configuration file. For information on the syntax of the entries, see "About the Payload Configuration File Syntax".

You use the fields in the event flist or storable class to define the business event. You need to specify the BRM events to publish, the data to publish, where to get the values for the fields, and how to present the data in the XML output.

For information on how to define events, see "Defining Business Events".

For an example of the relationship between the flist, event definition, and XML output, see "Event Flist, Event Definition, and XML Output Example".

If you need to publish valid XML documents for events, you must create data type definitions (DTDs) for the events. For a sample, see the DTDs for the default events in the *BRM_home***/sys/eai-js/dtds** directory.

Setting Up Multiple Publishers and Events

You can define multiple publishers in EAI Manager to publish separate sets of business events. You define a publisher by including a **<PublisherDefs>** tag for it in the **payloadconfig.xml** file. See "Publisher Definitions" for more information.

Using multiple publishers, you can ensure that applications receive only the events they need. For example, one publisher could publish the customer creation event to one application, while another publisher publishes the service creation event to another application.

Events can be published separately even if one event is part of another. For example, service creation can take place during customer creation, but the subscriber interested in service creation receives only that event, not the larger customer creation event.

Defining Business Events

To define business events:

1. In a text editor, open the payload configuration file (*BRM_home***/sys/eai_js/payloadconfig.xml**).
2. In the **<Publisher>** section, specify the business events you want to publish.

   For information on the syntax, see "Publisher Definitions".
3. (Optional) In the **<Publisher>** section, specify attributes for business events that will cause identifiers to be returned.

   See "Returning Identifiers from Enterprise Applications".
4. For each business event you specified, define the following attributes in the **<EventDefs>** section:
   - Source
   - Tag
   - StartEvent
   - (Optional) EndEvent

- (Optional) DataFrom
- (Optional) UseOnlyElement

5. (Optional) Include the following elements in the **<EventDefs>** section:
   - DTD
   - Attribute
   - Field
   - Search
   - SubElement
6. For each **SubElement** in the **<ElementDefs>** section, include the following attributes to define the sub-element:
   - Source
   - (Optional) DataFrom
   - Tag
7. (Optional) Include the following attributes:
   - UseOnlyElement
   - UseElementId
   - PinFld

   > **Note:**
   >
   > If you are using the flist (field list) output format, you *must* specify **PinFld**.

8. Include the following elements in the element definition:
   - Field
   - ExtendedInfo
9. (Optional) To check the configuration file for errors, run the **ValidateConfig** program.

   For more information, see "Validating Your Changes to the Payload Configuration File".
10. Save and close the file.
11. Ensure that the BRM events included in each newly defined business event are in your system's event notification list.

    See "Configuring EAI Manager".
12. Stop and restart the Payload Generator External Module (EM):

```
cd BRM_home/bin
pin_ctl bounce eai_js
```

> **Note:**
>
> If you rename the file, make sure you also change the file name in the Payload Generator EM **Infranet.properties** file. For more information, see "Configuring EAI Manager".

Removing Events That You Do Not Want to Publish

To remove events you do not want to publish:

1. In a text editor, open the payload configuration file (*BRM_home*/**sys/eai_js/payloadconfig.xml**).
2. Remove the events that you do not want to publish from the event list in the **<Publisher>** section.
3. (Optional) To check the configuration file for errors, run the **ValidateConfig** program.

   For more information, see "Validating Your Changes to the Payload Configuration File".
4. Save and close the file.
5. Stop and restart the Payload Generator EM:

```
cd BRM_home/bin
pin_ctl bounce eai_js
```

Returning Identifiers from Enterprise Applications

You can configure EAI Manager to receive identifiers returned from enterprise applications. For example, if data is published to a contact management application, that application may return user IDs. You can store those IDs in the BRM database.

You specify additional attributes in the **payloadconfig.xml** file when you define a business event that returns an identifier. These attributes are necessary only for business events that return an identifier.

See "Publisher Definitions" for more detailed information about the syntax.

In addition to including information about identifiers in the **<Publisher>** section of the **payloadconfig.xml** file, you must implement the **SetIdentifier** function in your connector application. See "SetIdentifier" for more information.

Changing the Format of Published Events

By default, EAI Manager publishes business events in XML format. You can publish events in flist format by editing the configuration file.

1. In a text editor, open the payload configuration file (*BRM_home*/**sys/eai_js/payloadconfig.xml**).
2. In the **<PublisherDefs>** section, change the value of **Format** from **XML** to **FLIST**:

```
<PublisherDefs>
    <Publisher DB="0.0.9.1" Format="FLIST">
```

3. (Optional) To check the configuration file for errors, run the **ValidateConfig** program.

   For more information, see "Validating Your Changes to the Payload Configuration File".
4. Save and close the file.
5. Stop and restart the Payload Generator EM:

```
cd BRM_home/bin
pin_ctl bounce eai_js
```

Validating Your Changes to the Payload Configuration File

After you edit the **payloadconfig.xml** configuration file, you can check the validity of the file by using the **ValidateConfig** program. This program checks the configuration file for errors and displays the list of events to be published along with a message that the file is valid. If there are errors, the **ValidateConfig** program specifies them.

To validate your changes to the configuration file, run the **ValidateConfig** program using the following syntax:

```
java com.portal.eai.ValidateConfig [config_file_name]
```

For example:

```
java com.portal.eai.ValidateConfig BRM_home/sys/eai_js/payloadconfig.xml
```

> **Note:**
>
> If the configuration file name is not specified, the **ValidateConfig** program uses the configuration file specified in the **Infranet.properties** file.

34
EAI DM Functions

Learn about the functions used by Oracle Communications Billing and Revenue Management (BRM) Enterprise Application Integration (EAI) Manager.

Topics in this document:

- AbortTransaction
- CommitTransaction
- FreeGlobalContext
- GetGlobalContext
- Initialize
- OpenTransaction
- PrepareCommit
- PublishEvent
- SetIdentifier
- Shutdown

AbortTransaction

This function is called by the EAI DM after the transaction is cancelled in BRM. For transactional systems, the transaction started with the **OpenTransaction** function needs to be rolled back with this function.

Syntax

```
int
AbortTransaction(
               void     *context);
```

Parameters

***context**
A pointer to the context returned by the **Initialize** function.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

CommitTransaction

This function is called by the EAI DM after BRM commits the transaction to its database. You commit the transaction opened to your system with this function.

Syntax

```
int
CommitTransaction(
                void      *context);
```

Parameters

context
A pointer to the context returned by the **Initialize** function.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

> **Note:**
>
> Failure to commit to your system does not cancel the transaction within BRM.

FreeGlobalContext

This function is called when the EAI DM process is shutting down. All resources allocated by the **InitializeGlobalContext** function are freed with this function.

Syntax

```
void
FreeGlobalContext(
                void      *gblContext);
```

Parameters

gblContext
A pointer to the global context initialized by **InitializeGlobalContext**.

Return Values

This function returns nothing.

GetGlobalContext

This function is called from a connector application to access the context initialized with **InitializeGlobalContext**.

Syntax

```
void
GetGlobalContext();
```

Parameters

This function has no parameters.

Return Values

This function returns nothing.

Initialize

This function is called by the EAI DM when it starts. This function performs all the initialization tasks such as resource allocations in this function implementation.

> **Note:**
>
> **Initialize** is called once for each EAI DM back end. The number of back ends is specified in the EAI DM **pin.conf** file.

Syntax

```
int
Initialize(
                void      **context,
                int        *output_type);
```

Parameters

****context***
A pointer to an open context. *context* is a transparent Binary Large Object (BLOB) or cookie that is passed to the connector application during subsequent calls. You need to manage memory in your application for the context.

****output_type***
Specifies the output format type: TYPE_XML or TYPE_FLIST. The format must match the value of the **Format** entry in the **payloadconfig.xml** file.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

OpenTransaction

This function is called by the EAI DM before it calls the **PublishEvent** function. Open a transaction to your system during this call.

Syntax

```
int
OpenTransaction(
                void      *context);
```

Parameters

****context***
A pointer to the context returned by the **Initialize** function. You can save any transaction-specific information in the context.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

PrepareCommit

This function is called by the EAI DM when BRM is about to commit the transaction to its database. If this function returns an error, the transaction within BRM is cancelled.

Syntax

```
int
PrepareCommit(
                void     *context);
```

Parameters

***context**
A pointer to the context returned by the **Initialize** function.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

PublishEvent

This function is called by the EAI DM when there is a business event to be published. There could be more than one **PublishEvent** call during a single transaction.

Syntax

```
int
PublishEvent(
                void     *context,
                void     *payload,
                char     *servicep);
```

Parameters

***context**
A pointer to the context returned by the **Initialize** function.

***payload**
A pointer to the event payload.

***servicep**
A pointer to the service that was used to log in when this business event was generated. You can use this parameter to identify duplicate logins.

Return Values

Returns **PIN_RESULT_PASS** if the operation is successful. Returns **PIN_RESULT_FAIL** if the operation fails.

SetIdentifier

This function is called from the connector application to set a return identifier for a published business event. If an identifier is set when the event is published, the DM returns the identifier when it sends a response flist.

Syntax

```
void
SetIdentifier(
                void     *identifier,
                int       idLen);
```

Parameters

identifier
A pointer to the identifier.

idLen
The length of the identifier.

Return Values

This function returns nothing.

Shutdown

This function is called by the EAI DM when it shuts down. You free all the resources allocated during initialization with this function.

> **Note:**
>
> To ensure an orderly shutdown and make sure that resources are reallocated, the connector application should send a SIGQUIT signal to the EAI_DM main process.

Syntax

```
void
Shutdown(
                void     *context);
```

Parameters

context
A pointer to the context returned by the **Initialize** function.

Return Values

This function returns nothing.

Part IV

Integrating BRM with an Apache Kafka Server

This part describes how to integrate and synchronize data between Oracle Communications Billing and Revenue Management (BRM) and an Apache Kafka server. It contains the following chapters:

- About Integrating BRM with an Apache Kafka Server
- Configuring BRM to Publish Notifications to Kafka Servers

35

About Integrating BRM with an Apache Kafka Server

Learn how to integrate Oracle Communications Billing and Revenue Management (BRM) and Apache Kafka servers by using the Kafka Data Manager (DM).

Topics in this document:

- About Integrating BRM with Kafka Servers
- About the EAI Framework for the Kafka DM
- About the CM and Notification Events
- About the Kafka DM

About Integrating BRM with Kafka Servers

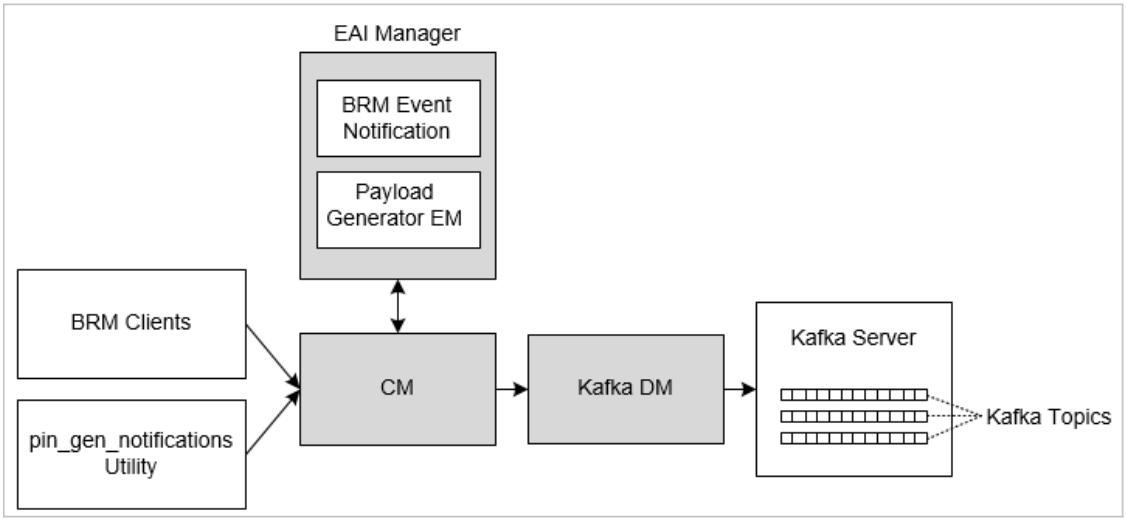
Integrating BRM with a Kafka server allows you to keep data synchronized between BRM and your external applications connected to the Kafka server. To synchronize data, BRM takes data from internal notification events and constructs a business event that is published to a topic in your Kafka server. Your external applications can then retrieve and process the data from the Kafka topic.

You integrate BRM with a Kafka server and configure it to publish data to a Kafka server by using the following BRM components:

- Connection Manager (CM)
- Enterprise Application Integration (EAI) framework, which consists of the event notification system and the Payload Generator External Module (EM).
- Kafka Data Manager (DM)

Figure 35-1 shows the BRM to Kafka server architecture and data flow.

Figure 35-1 BRM and Kafka Server Architecture



The data flow from BRM to your Kafka topics works as follows:

1. A notification event is generated in BRM when:
   - A customer's account is created or changed in a client application. For example, when a customer purchases a product.
   - The **pin_gen_notifications** utility runs. This utility creates notification events before or after a customer's balance expires, product expires, subscription is due for renewal, or bill is due. See "About Generating Notifications In Advance" and "About Generating Notifications After an Event Occurs" in *BRM Managing Customers* for more information.
2. The CM sends the event to the BRM event notification system.
3. The BRM event notification system sends the event to the Payload Generator EM.
4. The Payload Generator EM collects events in its cache until they compose a whole business event.
5. The Payload Generator EM generates the business event payload in flist format and then sends it to the CM.
6. Internally, the CM sends the business event to the PCM_OP_PUBLISH_EVENT opcode to enrich business events with subscriber preferences.
7. Internally, the CM sends the business event to the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode to perform any customizations on the event. By default, the policy opcode does not manipulate the data and returns the original input as output.
8. The CM sends the business event payload to the Kafka DM.
9. The Kafka DM transforms the business event payload from flist format into XML or JSON format. It then publishes the payload into one or more topics in your Kafka server.

Depending on the configuration, if the payload fails to publish successfully to the Kafka server, the Kafka DM either rolls back the transaction and returns an error to BRM or records the failed business event to a log file.

About the EAI Framework for the Kafka DM

You use the EAI framework to define business events for the Kafka server, to capture the BRM events that make up the business events, and to send completed business events to the Kafka DM.

The Kafka DM EAI framework consists of the following components:

- **BRM event notification**

  BRM event notification listens for events and, when they occur, calls the appropriate opcode. You specify the list of events that trigger an opcode call by editing an event notification file.

  The default Kafka DM event notification file specifies that when one of the events in **pin_notify_kafka_sync.xml** occurs to call an internal EAI framework publisher opcode (PCM_OP_PUBLISH_GEN_PAYLOAD) which, in turn, publishes the event to the Payload Generator EM.

  You can add or remove events from **pin_notify_kafka_sync.xml** file. See "Configuring Event Notification for Kafka Servers".

- **Payload Generator EM**

  The Payload Generator EM is responsible for collecting notification events until they form a complete business event, generating the business event, and then publishing it to the Kafka DM.

  You define which notification events the Payload Generator EM uses to form a complete business event for your Kafka server by using the Kafka DM payload file (**payloadconfig_kafka_sync.xml**). For example, the payload file specifies to collect the **/event/notification/rerating/start** and **/event/notification/rerating/end** events to form a complete Rerating business event. The default file includes definitions for business events such as AccountStatusUpdate, BillInfoUpdate, BillNow, CustCreate, ModifyBalanceGroup, and UpdateServices. You can modify the file by adding business events to it, removing default business events from it, or modifying the format in which the business events are published. For information about editing this file, see "Defining Business Events for Your Kafka Server".

Although the Kafka DM relies on the EAI framework, you do not need to install EAI Manager separately. All necessary EAI files are included with the Kafka DM components installed with BRM.

About the CM and Notification Events

When integrated with the Kafka DM, the CM is responsible for:

- Sending notification events to the EAI Framework.
- Enriching outgoing notification events with subscriber preferences or system preferences before they are sent to the Kafka DM. If configured to do so, the CM adds information such as the account's preferred language, delivery method, and time.

  The CM retrieves subscriber preferences from an account's **/profile/subscriber_preferences** object. If the object is missing or does not contain any preferences, it looks it up in the **/config/notification_spec** object.

  For more information, see "About Enriching Notifications with Additional Information" in *BRM Managing Customers*.

About the Kafka DM

The Kafka DM is responsible for sending BRM-generated business events to one or more topics in your Kafka server.

You can run the Kafka DM in one of these modes:

- **Asynchronous mode**: The Kafka DM records in a log file all business events that fail to publish to the Kafka server. You configure the name and location of the log file using the **<KafkaAsyncMode>** element in the *BRM_home***/sys/dm_kafka/log4j2.xml** file. Asynchronous mode is the default.
- **Synchronous mode**: When a business event fails to publish to the Kafka server, the Kafka DM rolls back the transaction and returns an error to BRM.

You define how the Kafka DM connects to your Kafka server and topics, the Kafka DM mode to use, which business events to publish to each Kafka topic, and the format and style of the payload by using the *BRM_home***/sys/dm_kafka/dm_kafka_config.xml** file. To configure the file, see "Mapping Business Events to Kafka Topics".

The default file creates a default Kafka topic named **BrmTopic** that accepts payloads in the XML format and ShortName style, but you can add topics, remove topics, modify the topic names, or modify the format and style accepted by the topics.

When the Kafka DM receives a business event payload from the Payload Generator EM, the Kafka DM converts the payload from flist format into XML or JSON format. It then determines whether the payload should be published to a Kafka topic by checking the **dm_kafka_config.xml** file. The entire contents of the business event are published to one or more Kafka topics.

36

Configuring BRM to Publish Notifications to Kafka Servers

Learn how to configure Oracle Communications Billing and Revenue Management (BRM) to publish notification events and messages to your Kafka server.

Topics in this document:

- Overview of BRM Configuration Tasks for Kafka Servers
- Installing the BRM Kafka DM
- Configuring Thread Pooling for the Kafka DM
- Enabling SSL between Kafka DM and Kafka Server
- Configuring Event Notification for Kafka Servers
- Defining Business Events for Your Kafka Server
- Mapping Business Events to Kafka Topics
- Configuring the Dynamic Key Value
- Configuring Where to Record Failed Events
- Customizing Notification Enrichment

Overview of BRM Configuration Tasks for Kafka Servers

The high-level tasks for configuring BRM to send notifications and messages to your Kafka server include:

1. Installing and setting up Apache Kafka and Apache ZooKeeper.

   For more information, see "Apache Kafka Quickstart" on the Apache Kafka web site.

2. Installing the BRM Kafka Data Manager (DM).

   See "Installing the BRM Kafka DM".

3. Setting up thread pooling for the Kafka DM by editing the Kafka DM's **Infranet.properties** file.

   See "Configuring Thread Pooling for the Kafka DM".

4. Enabling SSL between the Kafka DM and your Kafka Server.

   For more information, see "Enabling SSL between Kafka DM and Kafka Server".

5. Modifying the list of notification events to send to the Payload Generator External Module (EM) by editing the **pin_notify_kafka_sync** file.

   See "Configuring Event Notification for Kafka Servers".

6. Defining how the Payload Generator EM builds business events for your Kafka server by editing the **payloadconfig_kafka_sync.xml** file.

   See "Defining Business Events for Your Kafka Server".

7. Mapping business events to topics in your Kafka server by editing the **dm_kafka_config.xml** file.

   See "Mapping Business Events to Kafka Topics".

8. (Optional) Configuring BRM to replace the dynamic key in message payloads with the Kafka database number.

   See "Configuring the Dynamic Key Value".

9. (Optional) Setting the name and location of the file for logging failed business events.

   See "Configuring Where to Record Failed Events".

10. (Optional) Customizing the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode to modify notification events before they are sent to a Data Manager (DM).

    See "Customizing Notification Enrichment".

Installing the BRM Kafka DM

If you are installing BRM for the first time, you can install the Kafka DM by following these instructions:

1. Install the latest version of Apache Kafka. For instructions on downloading and installing Kafka, see "Apache Kafka Quickstart" on the Apache Kafka website.

   For the latest compatible software version, see "BRM Software Compatibility" in *BRM Compatibility Matrix*.

2. Set the Kafka environment variables:

   ```
   setenv KAFKA_HOME Kafka_path
   setenv KAFKA_BOOTSTRAP_SERVER_LIST KafkaHost1:port1,KafkaHost2:port2
   ```

   where:

   - *Kafka_path* is the path to the directory in which the Kafka library JARs are installed.
   - *KafkaHost1*:*port1*,*KafkaHost2*:*port2* are the hosts and ports that the Kafka client will connect to in a bootstrap Kafka cluster the first time it starts. You can specify any number of hosts and ports in this list.

     You can alternatively set this list in the **dm_kafka_config.xml**. See "Mapping Business Events to Kafka Topics".

3. Follow the instructions in "Installing BRM" in *BRM Installation Guide* to install the full build package. When the Installation Type screen appears during the installation process, do one of the following:

   - To install all BRM components, including the Kafka DM, select the **Complete** installation option.
   - To install the Kafka DM along with other individual BRM components, select the **Custom** installation option, select **Kafka Data Manager 15.0.0.*x*.0**, and then select any other optional managers that you want to install.

Configuring Thread Pooling for the Kafka DM

You configure how many parallel requests that the Kafka DM can receive from the CM by editing the *BRM_home*/**sys/dm_kafka/Infranet.properties** file. Under the file's **DM EAI Configuration** section, configure the entries listed in Table 36-1.

Table 36-1 Kafka DM Infranet.properties Entries

Entry	Description
infranet.dm.name	The name of the Kafka DM. If there are multiple Kafka DMs, each one must have a unique name. The default is **DM-KAFKA-1**.
infranet.server.enabletimeinfo	Whether to log the timing information for each opcode that is run within the Kafka DM. The valid values are **yes** and **no**. **Note**: Disable this entry for production systems.
infranet.connection.pool.enable	Whether to enable thread pooling for the Kafka DM: • **true**: Thread pooling is enabled. This is the default. • **false**: Thread pooling is disabled. In this case, a thread is spawned for each CM request.
infranet.connection.pool.size	The number of threads that can run in the JS server to accept requests from the CM. Enter a number from 1 through 2000. The default is **64**.

Enabling SSL between Kafka DM and Kafka Server

Apache Kafka allows Kafka clients, such as the Kafka DM, to use SSL for encrypting traffic and for authentication. By default, SSL is disabled between Kafka clients and the Kafka Server. To secure communications between them, you must enable SSL in both the Kafka Server and Kafka DM.

Enabling SSL in Kafka Server

To enable SSL in the Kafka Server:

1. Create a Kafka Server certificate, KeyStore, and TrustStore by following the instructions in "Encryption and Authentication using SSL" in the Apache Kafka documentation.
2. Update your Kafka **server.properties** file with the SSL parameters.
3. Verify that SSL is set up properly in the Kafka Server. To do so:
   a. Create a test Kafka client certificate, KeyStore, and TrustStore by following the instructions in "Encryption and Authentication using SSL" in the Apache Kafka documentation.
   b. Create a client **producer.properties** file for **kafka-console-producer** and then add the client SSL properties to it.
   c. Create a client **consumer.properties** file for **kafka-console-consumer** and then add the client SSL properties to it.
   d. In Terminal-1, run the Kafka Producer Console:

      ```
      kafka-console-producer.sh --broker-list domainName:portNumber --topic
      topicName --producer.config config/producer.properties
      ```

      where:

- *domainName* and *portNumber* are the domain name and port number for the system running the Kafka Server.
- *topicName* is the name of the topic in which to send messages.

e. In Terminal-2, run the Kafka Consumer Console:

```
kafka-console-consumer.sh --bootstrap-server domainName:portNumber --
topic topicName --from-beginning --consumer.config config/
consumer.properties
```

where:

- *domainName* and *portNumber* are the domain name and port number for the system running the Kafka Server Consumer.
- *topicName* is the name of the topic in which to retrieve messages.

f. Send sample messages from the Kafka Producer Console, and then verify that the messages are received in the Kafka Consumer Console.

Enabling SSL in Kafka DM

To enable SSL in the Kafka DM:

1. Create a Kafka client certificate, KeyStore, and TrustStore by following the instructions in "Encryption and Authentication using SSL" in the Apache Kafka documentation.
2. Verify that the client KeyStore and TrustStore are set up properly by running the following command:

   ```
   openssl s_client -debug -connect domainName:portNumber -tls1_2
   ```

   where *domainName* and *portNumber* are the domain name and port number for the system running the Kafka client.

   If the command's output does not display the certificate or if there are other error messages, the KeyStore is not set up properly.
3. Add all of the sensitive data, such as the KeyStore password and TrustStore password, as a key-value pair in the BRM wallet. To do so, use the Oracle **mkstore** utility.

   For example, to set the password for the TrustStore password, you could run this command:

   ```
   mkstore -wrl "/home/myuser/wallet" -createCredential "TrustStorePassword" "password"
   ```
4. Update the Kafka DM configuration file (**dm_kafka_config.xml**) with the SSL details. See "Mapping Business Events to Kafka Topics" for more information.

Configuring Event Notification for Kafka Servers

You define which notification events are sent to the Payload Generator EM by using the event notification file. The default Kafka DM event notification file (**pin_notify_kafka_sync**) specifies to send all account and pricing events to the Payload Generator EM, but you can add or exclude notification events from this file to accommodate your business needs.

To configure event notification for Kafka servers:

1. Open the *BRM_home***/sys/data/config/pin_notify_kafka_sync** file in an XML editor.

2. To add a notification event, use this syntax:

   ```
   opcode_number     flag     event
   ```

   where:

   - *opcode_number* is the number associated with the opcode to run when the event occurs. Opcode numbers are defined in header (***.h**) files in the *BRM_home***/include/ops** directory. To send the notification event to the Payload Generator EM, enter **1301** for the opcode number.
   - *flag* is the name of the flag to pass to the opcode when it is called by the event notification feature. **0** means no flag is passed.
   - *event* is the name of the event that triggers the execution of the opcode. You can use any BRM default or custom event defined in your system. Triggering events do not have to be persistent. For example, you can use notification events (see "About Notification Events") and events that you have excluded from the BRM database (see "Managing Database Usage" in *BRM System Administrator's Guide*).

   For example:

   ```
   1301     0     /event/session
   ```

   This example specifies that when an **/event/session** event occurs, BRM event notification calls opcode number 1301, which is the EAI framework publishing opcode (PCM_OP_PUBLISH_GEN_PAYLOAD). In this case, the contents of the event are passed to the opcode without any flags.

3. To exclude a notification event, comment it out by inserting a number sign (#) at the beginning of the entry. For example:

   ```
   # 1301     0     /event/session
   ```

4. Close and save the edited file.
5. If your system has multiple configuration files for event notification, merge them.

   Ensure that the merged file includes the contents from your *BRM_home***/sys/data/config/pin_notify_kafka_sync** file.

6. Load your final event notification list into the BRM database by using the **load_pin_notify** utility:

   ```
   load_pin_notify -v event_notification_file
   ```

   where *event_notification_file* is the path and name of the BRM event notification file.

Defining Business Events for Your Kafka Server

You define which notification events the Payload Generator EM uses to form a complete business event for your Kafka server by using the Kafka DM payload file (**payloadconfig_kafka_sync.xml**). You can add or remove business events from the file to accommodate your business needs.

For information, see "Defining Business Events".

Mapping Business Events to Kafka Topics

You specify how to connect the Kafka DM to your Kafka brokers, the Kafka DM mode to use, the authentication method for communication, and the Kafka topics to create by using the Kafka DM configuration file (*BRM_home*/**sys/dm_kafka/dm_kafka_config.xml**).

For each Kafka topic that you create, you specify:

- The name of the Kafka topic, which must be unique.
- The business events to include in the payload published to the Kafka topic.
- The payload format: XML or JSON. The Kafka DM converts the flists for each business event into the specified format.
- For the XML payload format, the style of all field names in the XML payload:
  - **ShortName**: The XML field names are in all capitals, such as <POID>, <ACCOUNT_OBJ>, and <SUBSCRIBER_PREFERENCES_INFO>. This is the default.
  - **CamelCase**: The XML field names are in CamelCase, such as <Poid>, <AccountObj>, and <SubscriberPreferencesInfo>.
  - **NewShortName**: The XML field names are in CamelCase and are prefixed with **fld**, such as <fldPoid>, <fldAccountObj>, and <fldString>.
  - **OC3CNotification**: The input is transformed to match the field and formatting requirements of Oracle Communications Convergent Charging Controller. Use this style if Convergent Charging Controller is your external notification application.
- (Optional) The key setting. See "About Setting Topic and Payload Keys".
- (Optional) The headers to add to each message sent to the topic. See "Adding Headers to Messages".
- (Optional) The payload settings. See "Adding Separate Payload Settings".
- (Optional) The mapping of BRM flist fields to XML or JSON elements. See "Mapping Flist Fields to Payload Tags".

For information about how to edit the **dm_kafka_config.xml** file, see "Editing the dm_kafka_config.xml File".

About Setting Topic and Payload Keys

> **Note:**
>
> Only XML version 2.0 of the **dm_kafka_config.xml** file supports topic and payload keys.

By default, each message or payload sent to a Kafka topic has its key set to the payload name, such as CustCreate. You can optionally change the key to another value by setting the **key** attribute.

The **key** attribute can be added to the **<DefaultTopicDefinition>**, **<TopicDefinition>**, and **<Payload>** elements, and it can be set to the following:

- **{PayloadName}**: This variable is replaced with the message payload name, such as CustDelete or CustCreate. This is the default.
- **{Random}**: This variable is replaced with a random number, such as 1644450215456.
- **{Dynamic}**: This variable is replaced with the value passed in the PCM_OP_PUBLISH_POL_PREP_EVENT output flist. See "Configuring the Dynamic Key Value" for more information.
- Static Value: The characters or numbers you enter will be passed literally. For example, if you enter **1.0.0.0**, the key will be set to **1.0.0.0**.

You can also combine attribute values, such as **{PayloadName}_1.0.0**.

The following shows sample entries for setting a key in the default BrmTopic topic and the NotificationTopic topic:

```
<DefaultTopicDefinition name="BrmTopic" format="XML" style="OC3CNotification"
key="A1"/>
<TopicDefinition name="NotificationTopic" format="XML"
style="OC3CNotification" key="{PayloadName}-1234">
```

In this example, messages sent to BrmTopic would have their key set to **A1**. Messages sent to NotificationTopic would have their key set to CustDelete-1234, CustCreate-1234, and so on.

For more information about adding keys to payloads, see "Adding Separate Payload Settings".

Adding Headers to Messages

> **Note:**
>
> Only XML version 2.0 of the **dm_kafka_config.xml** file supports adding headers to messages.

You can configure the Kafka DM to add one or more headers to each message sent to a specified Kafka topic. By default, headers are not added to messages, except when a topic's format is **XML** and style is **OC3CNotification**. For these topics, the default header is:

NOTIFICATION_TYPE=*PayloadName*, **NOTIFICATION_VERSION="1.0.0.0.0"**

Headers can be added under the **<DefaultTopicDefinition>** or **<TopicDefinition>** sections. To do so, add a **<Headers>** section. Under it, add a **<Header>** element for each header that you want to add:

```
<TopicDefinition name="MessageTopic" format="XML">
   <Headers>
      <Header key="key" value="value" style="style"/>
      <Header key="key" value="value" style="style"/>
   </Headers>
```

Table 36-2 describes the attributes in the **<Header>** element.

Table 36-2 Header Attributes

Attribute Name	Description
key	The header key, such as **NOTIFICATION_TAG** or **NOTIFICATION_VERSION**.
Value	The value for the header key, which can be the following: • **{PayloadName}**: This variable is replaced with the message payload name, such as CustDelete or CustCreate. • Static value: The characters or numbers you enter will be passed literally. For example, if you enter **1.0.0.0**, the header key value for all messages will be **1.0.0.0**. You can also combine both values, such as **{PayloadName}_2A**.
style	The style to use for **{PayloadName}** values: • **CamelCase**: The payload name is written in CamelCase, such as CustCreate or WelcomeMsg. • **UpperCaseUnderscore**: The payload name is written in all capitals with an underscore between each word, such as CUST_CREATE or WELCOME_MSG.

The following shows sample header entries for a topic where the header key is **NOTIFICATION_TAG**, the key value is **{PayloadName}_EVENT**, and the style is **UpperCaseUnderscore**:

```
<TopicDefinition name="MessageTopic" format="XML">
   <Headers>
      <Header key="NOTIFICATION_TAG" value="{PayloadName}_EVENT"
style="UpperCaseUnderscore"/>
   </Headers>
```

In this case, if the payload name is **BalanceExpiry**, the following header would be added to the message:

```
NOTIFICATION_TAG=BALANCE_EXPIRY_EVENT
```

Adding Separate Payload Settings

> **Note:**
>
> Only XML version 2.0 of the **dm_kafka_config.xml** file supports adding separate payloads.

For each topic, you can configure the Kafka DM to write business events in a separate message payload. To do so, you add a **<Payloads>** section under the **<DefaultTopicDefinition>** or **<TopicDefinition>** sections. Under **<Payloads>**, add a **<Payload>** element for each business event you want written to a separate message payload:

```
<Payloads>
   <Payload name="name" key="key" partition="partition">
```

```
   </Payload>
</Payloads>
```

Table 36-3 describes the attributes in the **<Payload>** element.

Table 36-3 Payload Attributes

Attribute Name	Description
name	The name of the business event to write to a separate payload.
key	The payload key setting: • **{PayloadName}**: This variable is replaced with the message payload name, such as CustDelete or CustCreate. This is the default. • **{Random}**: This variable is replaced with a random number, such as 1644450215456. • **{Dynamic}**: This variable is replaced with the value passed in the PCM_OP_PUBLISH_POL_PREP_EVENT output flist. See "Configuring the Dynamic Key Value" for more information. • Static Value: The characters or numbers you enter will be passed literally. For example, if you enter **1.0.0.0**, the key will be set to **1.0.0.0**. **Note**: Keys set at the payload level override the keys set at the topic level. See "About Setting Topic and Payload Keys" for more information
partition	The partition in which to write the message and topic. By default, the Kafka DM assigns messages to a random partition in the Kafka topic.

The following shows sample entries for the BalanceExpiry payload:

```
<Payloads>
   <Payload name="BalanceExpiry" key="ABCD" partition="1">
   </Payload>
</Payloads>
```

In this example, a **BalanceExpiry** business event would have its key set to **ABCD**, and would be loaded into partition 1 of the Kafka topic.

Mapping Flist Fields to Payload Tags

> **Note:**
>
> Only XML version 2.0 of the **dm_kafka_config.xml** file supports the mapping of flist fields.

The Kafka DM provides default mappings between a business event's flist fields and the XML or JSON elements in the payload sent to Kafka topics. You can override how the Kafka DM transforms one or more business event flist fields to XML or JSON elements at the topic level and the payload level.

> **Note:**
>
> If mappings for the same flist field are defined at both the topic level and the payload level, the mapping in the payload takes precedence.

Field mappings can be added under the **<DefaultTopicDefinition>**, **<TopicDefinition>**, and **<Payload>** sections. To do so, add a **<FieldMaps>** section. Under it, add a **<FieldMap>** element for each field that you want to override:

```
<TopicDefinition name="MessageTopic" format="XML">
   <FieldMaps>
      <FieldMap pinfld="field" tag="tag"/>
      <FieldMap pinfld="field" value="tag"/>
   </FieldMaps>
```

Table 36-4 describes the attributes in the **<FieldMap>** element.

Table 36-4 FieldMap Attributes

Attribute Name	Description
pinfld	The name of the flist field in the BRM business event payload.
tag	The name of the XML or JSON element in the payload published to the Kafka DM.

The following shows sample field mapping entries:

```
<TopicDefinition name="NotificationTopic" format="XML"
style="Notification">
   <FieldMaps>
      <FieldMap pinfld="PIN_FLD_ACCOUNT_OBJ" tag="AccountObjId1"/>
      <FieldMap pinfld="PIN_FLD_BAL_GRP_OBJ" tag="BalGrpPoidId1"/>
   </FieldMaps>
   <Payloads>
      <Payload name="BalanceExpiry">
         <FieldMaps>
            <FieldMap pinfld="PIN_FLD_ACCOUNT_OBJ" tag="AccountPoidId2"/>
         </FieldMaps>
      </Payload>
   </Payloads>
</TopicDefinition>
```

In this example, fields would be mapped as follows:

- For the **BalanceExpiry** payload name: PIN_FLD_ACCOUNT_OBJ is mapped to AccountPoidId2, PIN_FLD_BAL_GRP_OBJ is mapped to BalGrpPoidId1, and all other flist fields use the default mappings.
- All other payload names: PIN_FLD_ACCOUNT_OBJ is mapped to AccountObjId1, PIN_FLD_BAL_GRP_OBJ is mapped to BalGrpPoidId1, and all other flist fields use the default mappings.

Editing the dm_kafka_config.xml File

To map business events to Kafka topics:

1. Open the *BRM_home*/**sys/dm_kafka/dm_kafka_config.xml** file in an XML editor.
2. In the **<KafkaAsyncMode>** XML element, specify the Kafka DM mode to use:

   ```
   <KafkaAsyncMode>value</KafkaAsyncMode>
   ```

   where *value* is **true** for asynchronous mode, and **false** for synchronous mode. In asynchronous mode, the Kafka DM records in a log file all business events that fail to publish to the Kafka server. In synchronous mode, the Kafka DM returns errors to BRM when a business event fails to publish to the Kafka server.
3. In the following XML element, set the amount of time, in milliseconds, the Kafka DM waits for the Kafka server to respond:

   ```
   <ProducerConfig>max.block.ms=timeout</ProducerConfig>
   ```

   where *timeout* specifies the amount of time. If **<KafkaAsyncMode>** is set to **false**, set *timeout* to 3000 or higher. If **<KafkaAsyncMode>** is set to **true**, set *timeout* to 500 or less.
4. In the **<BootstrapServerList>** XML element, enter a comma-separated list of addresses for the Kafka brokers in this format:

   ```
   <BootstrapServerList>hostname1:port1,hostname2:port2</BootstrapServerList>
   ```

   You can alternatively leave the default value in the XML entry and set the list in the environment variable, as shown in "Installing the BRM Kafka DM."
5. To enable SSL or authentication between the Kafka DM and your Kafka server, do the following:
   a. Under the **<ProducerConfigs>** section, add **<ProducerConfig>** XML elements for your authentication protocol and mechanism. For information about the possible entries, see "Security" in the Apache Kafka documentation.
   b. For sensitive information such as passwords, create a token for the password value. For example:

      ```
      <ProducerConfig>ssl.truststore.password={TokenName}</ProducerConfig>
      ```

      (For XML version 2.0) Ensure that *TokenName* matches the key value stored in the Oracle wallet. See "Enabling SSL in Kafka DM".
   c. (For XML version 1.0 only) For each token, add an **<EncryptedVariable>** element that contains: the token name and the encrypted password.

      Use the **pin_crypt_app** utility to encrypt the password. See "pin_crypt_app" for more information.

      ```
      <EncryptedVariable>TokenName=&ozt|MyEncryptedPassword</
      EncryptedVariable>
      ```

This shows sample **<ProducerConfigs>** entries for authentication using the SASL_SSL protocol and SASL/PLAIN mechanism:

```
<ProducerConfigs>
   <ProducerConfig>ssl.truststore.location=${HOME}/kafka/keystores/
client.truststore.jks</ProducerConfig>
   <ProducerConfig>ssl.truststore.password={TrustStorePassword}</
ProducerConfig>
   <ProducerConfig>ssl.keystore.location=${HOME}/kafka/store/server/
server</ProducerConfig>
   <ProducerConfig>ssl.keystore.password={KeyStorePassword}</
ProducerConfig>
   <ProducerConfig>ssl.key.password={KeyPassword}</ProducerConfig>
   <ProducerConfig>security.protocol=SASL_SSL</ProducerConfig>
   <ProducerConfig>sasl.mechanism=PLAIN</ProducerConfig>

<ProducerConfig>sasl.jaas.config=org.apache.kafka.common.security.plain.Pla
inLoginModule required username="dmkafka" password="{Password}";</
ProducerConfig>
</ProducerConfigs>
```

6. In the **<DefaultTopicDefinition>** XML element, set the following:
   - **name** attribute: The name of the default Kafka topic.
   - **format** attribute: The payload format: **XML** or **JSON**.
   - **style** attribute: The style of XML payloads: **ShortName**, **CamelCase**, **NewShortName**, or **OC3CNotification**.
   - **key** attribute: The key to add to each message: **{PayloadName}**, **{Random}**, **{Dynamic}**, or a static value. This attribute is optional.
   - **<Headers>** element: Information about any headers to add to messages sent to this topic. This element is optional.
   - **<Payloads>** element: Information about any separate payloads to create in messages sent to this topic. This element is optional.
   - **<FieldsMaps>** element: Information about how to map specific flist fields to the target format. This element is optional.

   This example creates a default topic named **BrmTopic** that converts flists into XML payloads in the CamelCase style, adds a random key, adds the header **NOTIFICATION VERSION=1.0.0.0.0** to all messages sent to it, routes BalanceExpiry business events to partition 1, and maps the PIN_FLD_ACCOUNT_OBJ flist field.

```
<DefaultTopicDefinition name="BrmTopic" format="XML" style="CamelCase"
key="{Random}">
   <Headers>
      <Header key="NOTIFICATION_VERSION" value="1.0.0.0.0"/>
   </Headers>
   <Payloads>
      <Payload name="BalanceExpiry" key="ABCD" partition="1">
         <FieldMaps>
            <FieldMap pinfld="PIN_FLD_ACCOUNT_OBJ"
tag="AccountPoidId2"/>
         </FieldMaps>
      </Payload>
```

```
   </Payloads>
</DefaultTopicDefinition>
```

7. For each Kafka topic that you want to create, add a **<TopicDefinition>** XML element and set the following:
   - **name** attribute: The name of the Kafka topic
   - **format** attribute: The payload format: **XML** or **JSON**
   - **style** attribute: The style of XML payloads: **ShortName**, **CamelCase**, **NewShortName**, or **OC3CNotification**
   - **key** attribute: The key to add to each message: **{PayloadName}**, **{Random}**, **{Dynamic}**, or a static value. This attribute is optional.
   - **<Headers>** element: Information about any headers to add to messages sent to this topic. This element is optional.
   - **<Payloads>** element: Information about any separate payloads to create in messages sent to this topic. This element is optional.
   - **<FieldsMaps>** element: Information about how to map specific flist fields to the target format. This element is optional.

   For example:

```
<TopicDefinition name="NotificationTopic" format="XML"
style="OC3CNotification" key="{Dynamic}">
   <Headers>
      <Header key="NOTIFICATION_TAG" value="{PayloadName}Event"
style="CamelCase"/>
      <Header key="NOTIFICATION_VERSION" value="1.0.0.0.0"/>
   </Headers>
      <Payload name="BillDue" key="{PayloadName}"
partition="2">
         <FieldMaps>
            <FieldMap pinfld="PIN_FLD_ACCOUNT_OBJ"
tag="AccountPoidId1"/>
         </FieldMaps>
      </Payload>
</TopicDefinition>
```

8. Save and close the file.
9. Restart the Kafka DM for the changes to take effect.

Configuring the Dynamic Key Value

You can configure BRM to replace dynamic keys in message payloads with a value you specify in the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode.

To configure BRM to replace the dynamic key value:

1. Configure the **KafkaDBNumber** business parameter in the **notification** instance of the **/config/business_params** object:
   a. Use the following command to create an editable XML file from the **notification** instance of the **/config/business_params** object:

```
pin_bus_params -r BusParamsNotification bus_params_notification.xml
```

This command creates the XML file named **bus_params_notification.xml.out** in your working directory. If you do not want this file in your working directory, specify the path as part of the file name.

b. Set **KafkaDBNumber** to the Kafka database number:

```
<KafkaDBNumber>databaseNumber</KafkaDBNumber>
```

where *databaseNumber* is the number for the Kafka database. The default is **0.0.9.6 / 0**.

c. Save the file and change its name to **bus_params_notification.xml**.

d. Use the following command to load this change into the **/config/business_params** object:

```
pin_bus_params bus_params_notification.xml
```

You should run this command from the *BRM_home***/sys/data/config** directory, which includes support files used by the utility. To run it from a different directory, see "pin_bus_params".

e. Read the object with the **testnap** utility or the Object Browser to verify that all fields are correct.

For general instructions on using **testnap**, see "Using the testnap Utility to Test BRM". For information on how to use Object Browser, see "Reading Objects".

f. Stop and restart the CM.

For more information, see "Starting and Stopping the BRM System".

2. Customize the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode to compare the value of the PIN_FLD_POID input flist field to that of the **KafkaDBNumber** business parameter.

If they *match*, the policy opcode must pass the following output flist fields in the PIN_FLD_NOTIFICATION_KEY_INFO substruct:

- PIN_FLD_NOTIFICATION_KEY: Set this to your desired dynamic key value.
- PIN_FLD_STATUS: Set this to **1**.

For example, the following output flist settings specify to replace the dynamic key value with **ServiceLifeStateChangeExpiry**:

```
1 PIN_FLD_NOTIFICATION_KEY_INFO   SUBSTRUCT [0] allocated 20, used 5
2     PIN_FLD_NOTIFICATION_KEY          STR [0] "ServiceLifeStateChangeExpiry"
2     PIN_FLD_STATUS                   ENUM [0] 1
```

3. Set the message payload **key** attribute to **{Dynamic}**. See "About Setting Topic and Payload Keys".

Configuring Where to Record Failed Events

If you configure the Kafka DM to operate in Asynchronous mode, it records to a log file information about business events that fail to publish successfully to the Kafka server. Publishing might fail, for example, because the Kafka server is down or the connection fails.

> **Note:**
>
> Failed business events are written to a log file only in Asynchronous mode. When a business event fails to publish in Synchronous mode, the Kafka DM rolls back the transaction and returns an error to BRM.

By default, the Kafka DM records failed business events to the *BRM_log_file_home***/dm_kafka/kafka_failed_message.log** file, but you can change the name and location of the file.

To configure where the Kafka DM records failed business events, set the **filename** entry in the *BRM_home***/sys/dm_kafka/log4j2.xml** file:

```
<RollingFile name="KAFKA" fileName="${env:PIN_LOG}/dm_kafka/kafka_failed_message.log"
filePattern="${env:PIN_LOG}/dm_kafka/kafka_failed_message.log.%i">
```

The following shows a sample flist for a failed business event that could be recorded in the **kafka_failed_message.log** file:

```
0 PIN_FLD_EXTENDED_INFO        SUBSTRUCT [0] allocated 3, used 3
0 PIN_FLD_ACCOUNT_OBJ               POID [0] 0.0.0.1 /account 161798 0
0 PIN_FLD_LOGINS                   ARRAY [0] allocated 2, used 2
1     PIN_FLD_SERVICE_OBJ           POID [0] 0.0.0.1 /service/ip 162950 0
1     PIN_FLD_LOGIN                  STR [0] "user738"
0 PIN_FLD_LOGINS                   ARRAY [1] allocated 1, used 1
1     PIN_FLD_LOGIN                  STR [0] ""
0 PIN_FLD_LOGINS                   ARRAY [2] allocated 2, used 2
1     PIN_FLD_SERVICE_OBJ           POID [0] 0.0.0.1 /service/email 160518 0
1     PIN_FLD_LOGIN                  STR [0] "user738@portal.com"
0 PIN_FLD_LOGINS                   ARRAY [3] allocated 1, used 1
1     PIN_FLD_LOGIN                  STR [0] ""
0 PIN_FLD_STRING                     STR [0] "CustCreate"
0 PIN_FLD_END_T                   TSTAMP [0] (1669908080) 01/12/2022 07:21:20:000 AM
0 PIN_FLD_BAL_INFO                 ARRAY [0] allocated 3, used 3
1     PIN_FLD_OBJECT_CACHE_TYPE     ENUM [0] 0
1     PIN_FLD_BAL_GRP_OBJ           POID [0] 0.0.0.1 /balance_group 160006 1
1     PIN_FLD_BALANCES             ARRAY [840] allocated 1, used 1
2         PIN_FLD_CREDIT_PROFILE     INT [0] 3
0 PIN_FLD_CREATED_T               TSTAMP [0] (1672646104) 01/01/2023 23:55:04:692 PM
```

Customizing Notification Enrichment

By default, the PCM_OP_PUBLISH_EVENT opcode retrieves the delivery identifier for a delivery method, such as the email address for an email delivery method, by reading the **/profile/subscriber_preferences** object. If one or more delivery identifiers are not present in the object, the opcode includes in its output flist the PIN_FLD_NOTIFICATION_STATUS_INFO substruct indicating the list of missing delivery identifier tags. For example, this PIN_FLD_NOTIFICATION_STATUS_INFO substruct indicates that the Twitter handle and email address were not found in the **/profile/subscriber_preferences** object:

```
1  PIN_FLD_NOTIFICATION_STATUS_INFO SUBSTRUCT [0] allocated 20, used 2
2     PIN_FLD_STATUS      ENUM [0] 1
2     PIN_FLD_STATUSES   ARRAY [0] allocated 20, used 2
3        PIN_FLD_NAME      STR [0] "TwitterHandle"
2     PIN_FLD_STATUSES   ARRAY [1] allocated 20, used 2
3        PIN_FLD_NAME      STR [0] "Email"
```

You can customize the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode to look up or provide the missing delivery identifiers. The policy opcode can return a delivery identifier value in the PIN_FLD_VALUE output flist field of the PIN_FLD_NOTIFICATION_STATUS_INFO substruct. For example, this PIN_FLD_NOTIFICATION_STATUS_INFO substruct provides the **@SampleName** Twitter handle and **abcd@sample.com** email address:

```
1  PIN_FLD_NOTIFICATION_STATUS_INFO SUBSTRUCT [0] allocated 20, used 2
2     PIN_FLD_STATUS      ENUM [0] 1
2     PIN_FLD_STATUSES   ARRAY [0] allocated 20, used 2
3        PIN_FLD_NAME      STR [0] "TwitterHandle"
3        PIN_FLD_VALUE     STR [0] "@SampleName"
2     PIN_FLD_STATUSES   ARRAY [1] allocated 20, used 2
3        PIN_FLD_NAME      STR [0] "Email"
3        PIN_FLD_VALUE     STR [0] "abcd@sample.com"
```

If the policy opcode does not return a value in the PIN_FLD_VALUE field, the PCM_OP_PUBLISH_EVENT opcode returns an error and rolls back the entire transaction.

You can also customize the PCM_OP_PUBLISH_POL_PREP_EVENT policy opcode to not publish specific messages for specific publishers without having to roll back the entire transaction. To do so, customize the policy opcode to return in the PIN_FLD_NOTIFICATION_STATUS_INFO substruct, the PIN_FLD_STATUS output flist field set to one of the following:

- **0**: Indicates to not publish the message to this publisher. In this case, PCM_OP_PUBLISH_EVENT does not publish the event.
- **1**: Indicates that the message needs to be published to this publisher. In this case, PCM_OP_PUBLISH_EVENT returns an error and rolls back the entire transaction.

For example, this specifies to not publish the message to this publisher:

```
1  PIN_FLD_NOTIFICATION_STATUS_INFO SUBSTRUCT [0] allocated 20, used 2
2     PIN_FLD_STATUS  ENUM [0] 0
```

Part V

Creating and Customizing Client Applications

This part describes how to create and customize Oracle Communications Billing and Revenue Management (BRM) client applications. It contains the following chapters:

- Adding New Client Applications
- Using Transactions in Your Client Application
- Adding or Changing Login Options
- Creating Applications that Run on Multischema Systems
- Creating BRM Client Applications by Using the MTA Framework
- Creating Client Applications by Using Java PCM
- Creating Client Applications by Using Perl PCM
- Creating Client Applications by Using PCM C++

37

Adding New Client Applications

Learn how to add new client applications and how those applications function within the Oracle Communications Billing and Revenue Management (BRM) system.

Topics in this document:

- About Adding New Client Applications
- Implementing Timeout for Requests in Your Application
- Configuring Your Custom Application
- Creating a Client Application in C
- Using the Sample Applications
- About Adding Virtual Column Support to Your Applications

About Adding New Client Applications

Client applications can be virtually any type of program, including GUI-based tools, Web-based tools, network-enabled applications, batch jobs, cron jobs, and so on. You can write custom application programs to create, manipulate, delete, and display custom objects, which in turn implement a business policy.

The most common custom application captures external events of some type; for example, number of bytes downloaded from a web page. These events are then submitted to BRM, details of the event stored, and the event charges assessed if necessary.

BRM includes a set of client libraries that make it easier to create custom applications. Existing BRM applications use these client libraries. For more information about the libraries, see "About the BRM Client Access Libraries".

You can use the following options when you add custom applications to BRM:

- Using Existing System Opcodes
- Using Custom Opcodes
- Using a Custom Data Manager (DM)

Using Existing System Opcodes

You use the base opcodes to create, read, write, delete, and search objects. The base opcodes also provide programmatic access to transaction commands. More complex opcodes are implemented by the Facilities Modules (FMs). You use the policy opcodes to implement business decisions. These higher-level opcodes are translated by the FMs to base opcodes and sent to Data Managers (DMs) for processing.

In a client application, you use opcodes to record events such as purchasing a charge offer, changing a credit limit, creating an account or service, changing customer information, verifying a password, and looking up names and addresses.

In your application, you can call any of the BRM opcodes without changing the BRM system. You need to determine when to use a particular system opcode, what constitutes an event, and then call the appropriate opcode in your application.

Your application must include the header files corresponding to the opcodes you use. All FM opcodes include the header file for base opcodes, so you do not need to include it unless your application uses only base opcodes.

Using Custom Opcodes

If the system opcodes do not provide the required functionality, you can create custom opcodes. However, you must implement your new opcodes in a custom FM and configure the custom FM with each Connection Manager (CM).

For the application to communicate with the custom FM, the application and the FM have to agree on the opcodes to pass and the contents of their input and output flists, error buffers, and flags. The custom FM translates the new opcodes into base opcodes. Then, it can send the opcodes directly to a DM, or it can call other opcodes implemented by the standard FMs.

You call the custom opcodes in the same way with the same API by using PCM_OP() as the system opcode. Therefore, you must supply all the parameters, just as in the case of a system opcode.

Create a header file in which you define your custom opcodes. Include the header file in both the application and the custom FM source files. Compile the application and the custom FM with the new header file.

For examples of including opcodes in the header file, see the header files (**.h**) in the *BRM_home***/include** directory, where *BRM_home* is the directory in which the BRM server software is installed.

> **Tip:**
>
> Defining your custom opcodes in a separate **.h** file helps you avoid updating FM header files when you upgrade to a new release of BRM.

Using a Custom Data Manager (DM)

Your application can communicate with a custom DM if the custom DM and the application agree on the semantics of the input and return flists.

You attach fields to the input flist that have meaning to the custom DM. These new fields act as opcodes to the custom DM. The return flist is the mirror image of the input flist in the sense that the application and custom storage manager agree on the meaning of the field-value pairs returned by the storage manager. The storage manager is responsible for setting error buffer values.

You create new fields with the PIN_MAKE_FLD macro. For information on the PIN_MAKE_FLD macro, see **pcm.h** in the *BRM_home***/include** directory for field value ranges and examples of PIN_MAKE_FLD.

> **Tip:**
>
> To avoid updating the Portal **.h** files every time you upgrade BRM, define your custom fields in a separate header file and include that file in your application.

Implementing Timeout for Requests in Your Application

You can specify a timeout value for each connection to the CM. This enables you to set different timeout values for different operations. For example, you can set different timeout values for authorization and stop-accounting requests, or you can dynamically increase or decrease the timeout value for different operations based on the system load.

To specify a timeout value in milliseconds for a connection, pass the PIN_FLD_TIMEOUT_IN_MS field in the input flist to the PCM_CONTEXT_OPEN function. The PIN_FLD_TIMEOUT_IN_MS value is applicable after the client connects to the server. Before the connection occurs, this setting is not effective. It does not report a timeout if the client cannot connect to the server at all.

The timeout value you specify applies to all the opcodes called during that open session and overrides the value in the client configuration file. You must ensure that your client application handles the timeout, closes its connection to the CM by calling PCM_CONTEXT_CLOSE, and cleans up the transaction context.

> **Note:**
>
> When the timeout occurs, the CM does not provide any feedback about the success or failure of the request it received. When the CM detects the closed connection, it rolls back the ongoing transaction and shuts down.

Configuring Your Custom Application

You must set up the following applications to access the same database schema:

- The client application
- At least one CM
- At least one DM

The client application makes the connection by using the BRM database number in all three configuration files: the application's, the CM's, and the DM's. The database number is arbitrary, but it is determined before the system is installed. After the system is installed, you cannot change this number because it is encoded in every object in the database schema.

The client application must use POIDs with the correct database number. The system routes object requests on the basis of POIDs, which include the database number.

In your custom application, use the database number returned by PCM_CONNECT(). If you are using PCM_CONTEXT_OPEN(), call **PCM_GET_USERID()** and then PIN_POID_GET_DB() on the POID.

> **Caution:**
>
> Do not get the database number or the userid POID from the configuration file by calling **pin_conf()** in your application.

Creating a Client Application in C

You write client applications by using the PCM opcodes, which send and receive flists to the BRM database. Each opcode has a corresponding input and return flist.

Flists are used to hold return values for two important reasons:

- The macro call itself does not return a value.
- An flist can contain an arbitrary number of fields and values that is frequently not known in advance.

Your custom applications must include the header files that correspond to the FM opcodes you use. Which file to include depends on which opcodes you use. The header file for base opcodes needs to be included if you are using only base opcodes, which is unlikely.

You use the PCM_OP() macro to pass PCM opcodes and flists to BRM. The system returns an flist. You create input flists for the call to PCM_OP() and routine returns the results in an flist. You use the return flists and then destroy them.

The following pseudo-code shows the format of most client programs:

```
#include "pcm.h"
/* header file corresponding to the FM opcode you're using */
#include "ops/file.h"
#include "pin_errs.h"
    main()
     .
     .
     .
         /* open a database context */
         PCM_CONTEXT_OPEN()

        /* clear error buffer */
         PIN_ERRBUF_CLEAR(&ebuf);

        /* send opcode to system, based on user activity or application
            function. */

PCM_OP(input_flist, opcode, return_flist, &ebuf)

        /* check for errors */
         if (PIN_ERRBUF_IS_ERR(&ebuf)) {
             /* handle error */
         } else {
             /* ok - no errors */
         }
         .
         .
         .

        /* close database context */
         PCM_CONTEXT_CLOSE()
     .
```

```
.
.
exit(0);
```

Compiling and Linking Your Programs

You do not have to follow any special precompilation or other steps to compile and link applications. Both static and dynamic versions of BRM libraries are provided.

Linux client libraries are multi-thread safe.

To compile and link your application:

1. Compile using the **include** files in the *BRM_SDK_home***/include** directory.
2. Link to the libraries in *BRM_SDK_home***/lib**.

   See the sample applications and their make files for more information.

Table 37-1 lists the supported compilers:

Table 37-1 Supported Compilers

Operating System	Compiler
Linux	gcc compiler

Guidelines for Developing Applications in C on Linux Platforms

Follow these guidelines to develop custom applications in C on Linux:

- Include the appropriate library file at link time: **libportal.so** for Linux
- Add *BRM_SDK_home***/include** to the list of **include** file directories to search.
- In the preprocessor directives, be sure to include the following symbol:

  ```
  PIN_USE_ANSI_HDRS.
  ```

Using the Sample Applications

BRM SDK includes sample applications and code and source code for policy FMs that you can refer to for coding examples.

Sample Applications

BRM SDK includes sample applications and code in C, C++, Java, and Perl. For a complete list of the sample applications, see "Sample Applications" in *BRM Developer's Reference*.

Before you write your program, try compiling and linking copies of these programs to familiarize yourself with the system. These programs are located in *BRM_SDK_home***/source/samples**. This directory also includes a sample application configuration file.

> **Caution:**
>
> Do not run the sample programs on a production system. Some programs fill the database with test objects. Remove the test objects before building your production system.

Policy FM Source Files

BRM SDK includes the source code for all the policy opcodes. You can refer to them for BRM coding examples. You can find the Customer Policy FM opcode source files in *BRM_SDK_home*/**source/sys**. Each policy FM has its own directory containing the source files for the included opcodes and a make file and other support files.

About Adding Virtual Column Support to Your Applications

This section explains the programming considerations of creating an application to work with BRM virtual columns and applies to custom applications that interact with the BRM database directly. For information about using virtual columns in the BRM database, see the discussion on generating virtual columns in *BRM System Administrator's Guide*.

> **Caution:**
>
> - Always use the BRM API to manipulate data. Changing data in the database without using the API can corrupt the data.
> - Do not use SQL commands to change data in the database. Always use the API.

Custom applications can perform read operations on virtual columns but cannot perform update or insert operations. The values of virtual columns are computed dynamically, and attempts to modify them directly result in an error.

BRM creates virtual columns for the POID *field_name*_**type** columns on event tables in the BRM database. If your custom applications must update or insert data in these physical columns after they have been converted to virtual columns, you must make your applications interact with the virtual columns' respective supporting column.

Each BRM virtual column is associated with a supporting column that stores the storable class ID. The supporting columns can be modified and use the suffix *field_name*_**type_id** (the virtual columns use the suffix *field_name*_**type**).

The following examples demonstrate how custom applications can perform update and insert operations on the supporting columns of physical columns that have become virtual-column enabled.

> **Note:**
>
> The **get_object_id** function shown in the examples is available in the PIN_VIRTUAL_COLUMNS package.

Consider a table **event_t** with virtual column **session_obj_type**. The virtual column has a **session_obj_type_id** supporting column, which stores the ID corresponding to the type value of the virtual column.

- **Update operation example**

  Any custom application/PL/SQL updating the column **session_obj_type** using SQL

  ```
  update event_t set session_obj_type = '/service/telco';
  ```

  will have to be modified to

  ```
  update event_t set session_obj_type_id = pin_virtual_column.get_object_id('/service/
  telco');
  ```

- **Insert operation example**

  Any custom application/PL/SQL inserting values into column **session_obj_type** with SQL

  ```
  insert into event_t (poid_type) values (pin_virtual_columns.get_object_id('/event'));
  ```

  will have to be modified to

  ```
  insert into event_t (poid_type_id) values (pin_virtual_columns.get_object_id('/
  event'));
  ```

38

Using Transactions in Your Client Application

Learn how to manage transactions in custom Oracle Communications Billing and Revenue Management (BRM) client applications.

Topics in this document:

- Using Transactions
- Types of Transactions
- About Committing Transactions
- About Cancelling Transactions
- About the Transaction Base Opcodes

Using Transactions

Transactions enable an application to perform operations on multiple objects as if they were run simultaneously. This guarantees the integrity of the data when related changes need to be made to a set of objects.

In your application, you can call PCM_OP_TRANS_OPEN before calling an opcode and PCM_OP_TRANS_ABORT or PCM_OP_TRANS_COMMIT after the opcode calls.

Only one transaction at a time can be opened on a PCM context. A transaction is opened on a specific database schema specified by the POID database number in the input flist. All operations performed in an open transaction must manipulate data within the same database.

Any changes made within an open transaction can be cancelled at any time and all changes are completely erased. These actions cancel an open transaction:

- You use the PCM_OP_TRANS_ABORT opcode.
- The application exits or closes the PCM context.
- A system error occurs and connectivity is lost between the application and the database.

The system tracks the transaction along with the context argument used by most of the PCM Library macros. If the context pointer passed has an outstanding transaction, it is used automatically.

Keeping a transaction open for a long time can affect performance because the system maintains a frozen view of the data while changes are made by other applications. It is not recommended that you leave transactions open while long-latency tasks, such as prompting a user for input, are performed.

In general, any PCM opcode can be run within an open transaction, and its effect follows the transactional rules. However, some Facilities Module opcodes that interface to legacy systems or external systems do not follow the transactional rules (that is, they can't be undone). Opcodes with this limitation must check for an open transaction and return an error if an application attempts to run the opcode within the open transaction.

Types of Transactions

When you use the PCM_OP_TRANS_OPEN opcode to open a transaction, you can use the following flags to open different types of transactions:

- PCM_TRANS_OPEN_READONLY. See "Read-Only Transactions".
- PCM_TRANS_OPEN_READWRITE. See "Read-Write Transactions".
- PCM_TRANS_OPEN_LOCK_OBJ. See "Transaction with a Locked Objects".
- PCM_TRANS_OPEN_LOCK_DEFAULT. See "Transaction with a Locked Default Balance Group".

> **Note:**
>
> For J2EE-compliant applications, use JCA Resource Adapter to open extended architecture (XA) transactions through the XAResource interface. For more information, see "About BRM JCA Resource Adapter Transaction Management" see in *BRM JCA Resource Adapter*.

Read-Only Transactions

Use the PCM_TRANS_OPEN_READONLY flag to open a read-only transaction.

Use this type if operations will not change any data in the transaction.

From the application's point of view, a read-only transaction freezes the data in the database. The application does not see any changes to data made by other applications while the transaction is open. This allows data to be examined in a series of operations without being changed in mid-process.

Read-only transactions are more efficient and should be used when possible. Any number of read-only transactions can be open against a database at once.

Read-Write Transactions

Use the PCM_TRANS_OPEN_READWRITE flag to open a read-write transaction.

A read-write transaction freezes the data in the database from the application's point of view, and allows changes to be made to the data set. These changes are not seen by any other application until the transaction is committed. This allows the effects of a series of operations performed on objects to occur simultaneously when the transaction is committed.

Any number of read-write transactions can be open against a database at once.

Transaction with a Locked Objects

Use the PCM_TRANS_OPEN_LOCK_OBJ flag to open a transaction and lock an object as part of the transaction.

A lock-object transaction is useful when two applications must synchronize the operations they perform on the same object. Lock-object transactions are the same as read-write transactions,

with the addition of the object lock. If you use a lock-object transaction, you must specify the PCM_TRANS_OPEN_READWRITE flag.

If an application tries to open a lock-object transaction on an object that is already locked by another application, it will be held off until the application that currently holds the object finishes its transaction and unlocks the object.

Transaction with a Locked Default Balance Group

Use the PCM_TRANS_OPEN_LOCK_DEFAULT flag to open a transaction and lock the default balance group object only as part of the transaction.

Most opcode transactions lock the account object, if used, at the beginning of a transaction. This provides reliable data consistency but in systems that use account hierarchies, it can also cause a lot of serialization which decreases the throughput of the system. You can use the PCM_TRANS_OPEN_LOCK_DEFAULT flag to open a transaction that locks only the default balance group for the account instead of the sum of all the account objects in the hierarchy. See "Locking Specific Objects".

If you use a lock default balance group transaction, you must specify the PCM_TRANS_OPEN_READWRITE flag and not specify the PCM_TRANS_OPEN_LOCK_OBJ flag.

If an application tries to open a transaction on a balance group that is already locked by another application, it will be held off until the application that currently holds the object finishes its transaction and unlocks the object.

About Committing Transactions

Changes made within an open transaction are not permanent or visible to other applications until the transaction has been successfully committed.

Committing a transaction has these effects:

- The transaction is closed and all data changes made within the open transaction take effect in the data set. The changes become visible to all other applications (subject to their open transactions).
- The application's view of the data set is no longer frozen in time, so changes made by other applications are now visible to the application.
- If an object was locked, it is unlocked.
- The application is free to open another transaction. Subsequent operations on the PCM context are unrelated to the closed transaction.

> **Note:**
>
> For J2EE-compliant applications, use JCA Resource Adapter to commit XA transactions through the XAResource interface. The adapter supports both single-phase and two-phase commits. For more information, see "About BRM JCA Resource Adapter Transaction Management" in *BRM JCA Resource Adapter*.

About Cancelling Transactions

Cancelling a transaction has the following effects:

- All data changes made within the open transaction are discarded, so no data is changed by operations related to the transaction.
- If an object was locked, it is unlocked.
- The transaction is closed, and subsequent operations on the PCM context are unrelated to the closed transaction. The application is free to open another transaction.
- The application's view of the data set is no longer frozen in time, so changes made by other applications are visible to the application.

> **Note:**
>
> For J2EE-compliant applications, use JCA Resource Adapter to roll back XA transactions through the XAResource interface. For more information, see "About BRM JCA Resource Adapter Transaction Management" in *BRM JCA Resource Adapter*.

About the Transaction Base Opcodes

Use the following opcodes to manage transactions:

- To open transactions, use PCM_OP_TRANS_OPEN.
- To commit transaction, use PCM_OP_TRANS_COMMIT.
- To cancel transactions, use PCM_OP_TRANS_ABORT.

Customizing How to Open Transactions

To customize how to open transactions, use PCM_OP_TRANS_POL_OPEN.

This opcode gets the same flist that PCM_OP_TRANS_OPEN does. The return flist then becomes the transaction ID flist; it can contain whatever you want to put in it. That flist then becomes the input to PCM_OP_TRANS_POL_COMMIT and PCM_OP_TRANS_POL_ABORT. The return flists from those opcodes are ignored.

Customizing the Verification Process for Committing a Transaction Opcode

To customize how to verify the readiness of an external system to commit a transaction opcode, use PCM_OP_TRANS_POL_PREP_COMMIT.

This opcode provides BRM with preparatory notice of a pending commit process for transaction policies working with an external system. This is its overall process:

1. Open a transaction in each system.
2. Do the work authorized by the transaction.
3. Verify that the external system will be able to commit the transaction.
4. Commit the transaction in BRM.

5. Commit the transaction in the external system.

PCM_OP_TRANS_POL_PREP_COMMIT verifies that the external system will be able to commit the transaction. If the transaction is successfully committed, the CM calls PCM_OP_TRANS_COMMIT, and upon a successful commit transaction of that opcode it calls PCM_OP_TRANS_POL_COMMIT.

If PCM_OP_TRANS_POL_PREP_COMMIT fails, the CM automatically cancels the transaction using PCM_OP_TRANS_ABORT and PCM_OP_TRANS_POL_ABORT.

Customizing How to Commit a Transaction

To customize how to commit a transaction, use PCM_OP_TRANS_POL_COMMIT.

The return flist from PCM_OP_TRANS_POL_OPEN becomes the transaction ID flist; it can contain whatever you want to put in it. That flist then becomes the input to PCM_OP_TRANS_POL_COMMIT. The return flist from this opcode is ignored.

Customizing How to Cancel Transactions

To customize how to cancel transactions, use PCM_OP_TRANS_POL_ABORT.

The return flist from PCM_OP_TRANS_POL_OPEN becomes the transaction ID flist; it can contain whatever you want to put in it. That flist then becomes the input to PCM_OP_TRANS_POL_ABORT. The return flist from this opcode is ignored.

39

Adding or Changing Login Options

Learn how to create a login and password for custom applications to access Oracle Communications Billing and Revenue Management (BRM).

Topics in this document:

- About Customizing the Login Account for Your Application
- Creating Several admin_client Services with Different Permissions

About Customizing the Login Account for Your Application

You can use the default root account to log in. However, to properly manage access and permissions to BRM, you must create a BRM account for each custom application that you create.

To change the default login for your application, perform the following tasks:

1. Use Billing Care to create a BRM account with **pcm_client** service for your application.

   See "Creating an Account for Your Application".

   You can create an account for each instance of the application to manage permissions to a fine detail of control.
2. Provide the login and password to the application at runtime. See "Providing Login and Password to Your Custom Application" for instructions.

Creating an Account for Your Application

To prevent unwanted billing, the account that owns your custom **/service/pcm_client** and **/service/admin_client services** must be nonbilling. You create a nonbilling account by specifying the accounting type to PIN_BILL_TYPE_UNDEFINED.

To set up a nonbilling account:

1. Create an **/account** object.

   For information on creating objects, see "Creating Custom Fields and Storable Classes".
2. Change its PIN_FLD_BILL_TYPE value to PIN_BILL_TYPE_UNDEFINED.
3. Use Billing Care to create an account with **service/pcm_client** for your custom application.

Providing Login and Password to Your Custom Application

You can use one of the following methods to pass the login and password to your application:

- You can have the application user enter the login and password at runtime. This is the most secure way because there are no configuration files to be read.

  To use this method, call PCM_CONTEXT_OPEN in your application and build a login flist.

- You can get the login name and password from the application configuration file. This method allows the application to start automatically and reconnect. However, you must secure the configuration file to prevent unauthorized access.

  To use this method, call PCM_CONNECT in your application to open a PCM context.

  This routine reads the login type, name and password entries from your application configuration file. It then calls PCM_CONTEXT_OPEN with an input flist containing values for login type, name, and password from the configuration file.

  For an example of how to use this routine, see **sample_app.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Configuring System Passwords

After you create the new service and account for your application, edit the **userid** entry in your application configuration file to point to your new service.

You can specify that your application requires a login name and password to connect to BRM by setting **login_type** to **1** in the login information section of your application's configuration file.

In your application's configuration file, include entries for login type, name, and password using this syntax:

```
- nap   login_type    login_type
- nap   login_name    login_name
- nap   login_pw      password
```

For example:

```
- nap   login_type    1
- nap   login_name    Portal_user
- nap   login_pw      password
```

Creating Several admin_client Services with Different Permissions

You can create several admin_client services with different permissions to manage access and permissions to BRM components. Permissions are stored in the **/service/admin_client** object in the PIN_FLD_PERMITTEDS array.

1. Create the **/service/admin_client** objects that are owned by several accounts.

   For information on creating objects, see "Creating Custom Fields and Storable Classes".

2. Add as many permissions to the service permissions array (PIN_FLD_PERMITTEDS) as you want.

   For information about the array's format, see the **/service/admin_client** storable class specification.

40

Creating Applications that Run on Multischema Systems

Learn how to create custom applications that run on an Oracle Communications Billing and Revenue Management (BRM) multischema system.

Topics in this document:

- About Working with Multiple Schemas
- Creating Accounts in a Multischema System
- Maintaining Transactional Integrity
- Searching for Accounts across Database Schemas
- Finding How Many Database Schemas You Have
- Bill Numbering

About Working with Multiple Schemas

Generally, making applications work with multiple database schemas is not all that different from making them work with a single schema.

Accounts are distributed across schemas, but applications log in to the correct schema for an account based on the login name and service type. When an application logs in to BRM, it gets the schema context for the account it logged in as. An event for the login session for that application is created in the schema that hosts the account.

After the application has logged in, it has access to the entire BRM database for reads and writes on all storable classes that are modifiable. In most cases, after an account context is established, all subsequent operations for the account are performed in the single schema where the context was opened.

Creating Accounts in a Multischema System

Use the PCM_OP_CUST_COMMIT_CUSTOMER opcode to create accounts just as you would for a single-schema system. The opcode uses the **/config/distribution** object created by using the **load_config_dist** utility to determine which schema your account is created in.

You can specify which schema new accounts should be created in by editing the **config_dist.conf** configuration file. For more information, see "Setting Database Priorities" in *BRM System Administrator's Guide*.

> **Note:**
>
> Billing groups must reside in the same schema.

Maintaining Transactional Integrity

> **Note:**
>
> Remember, after you find an account to modify data in, confine all operations possible to that schema.

Although an application can connect to multiple database schemas and manipulate data in any schema, a transaction can only manipulate data in a single schema. To perform a transaction on more than one schema, you must close the existing transaction, open a context to the other schema, and open another transaction. An application that needs to perform the same operation on all accounts (such as billing or invoicing) should be run as a separate instance in each schema.

You must use the database number returned by PCM_CONNECT or PCM_CONTEXT_OPEN for all transactions within the context you open. These opcodes pass in an account's user name and return the database number for that account. To prevent losing transactional integrity, avoid opening contexts to multiple schemas whenever possible.

The exception to this rule is the rare occasion when you need to access information in any of the pricing storable classes. Embedded in these storable classes is the account information (including database number) of the person who changed that information. All account references are exact references. Managing this information can require you to switch contexts to another schema with a new call to PCM_CONNECT or PCM_CONTEXT_OPEN.

Searching for Accounts across Database Schemas

This section describes how to search for accounts across database schemas.

You can use the following opcodes to find a single account:

- Use the PCM_OP_ACT_FIND opcode to find an account based on the login and service type. This opcode finds and returns the account POID (including the correct database number) of a single account.
- Use the PCM_OP_GLOBAL_SEARCH opcode to find an account based on other account attributes. This opcode returns any fields that you specify on the input flist.

To find POIDS of multiple accounts across multiple database schemas, use the PCM_OP_GLOBAL_SEARCH opcodes. The global search opcodes can also be used to search for a set of objects that reside in multiple database schemas (for example, all events from a particular day). See "Searching for Objects in the BRM Database" for a complete discussion of searching for accounts across multiple database schemas.

> **Note:**
>
> Remember to use nonglobal searches for better performance whenever possible. After you get the results of a global search, you can improve your application's overall performance by dividing the database read and write operations among database schemas.

Finding How Many Database Schemas You Have

Use the **testnap** utility to find the number of database schemas connected to your BRM system. The following example shows **testnap** being started and then displays the contents of an flist named **1**. This flist is designed to match all **root** accounts. In the next step, this flist is passed to PCM_OP_GLOBAL_SEARCH (opcode number **25**), which searches all database schemas. In the final step, **testnap** searches all database schemas for their **root** accounts. Each database schema has only one **root** account (in the **/service** storable class), so the result of this search is a listing of all the database schemas currently connected to your BRM system. In this example, there are two: 0.0.0.1 and 0.0.0.2.

```
testnap
input flist:

d 1
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_FLAGS           INT [0] 0
0 PIN_FLD_TEMPLATE        STR [0] "select X from /service
where F1 like V1 and F2 = V2 "
0 PIN_FLD_ARGS          ARRAY [1]
1     PIN_FLD_LOGIN           STR [0] "root.0.0.0%"
0 PIN_FLD_ARGS          ARRAY [2]
1     PIN_FLD_POID          POID [0] 0.0.0.0 /service/pcm_client -1 0
0 PIN_FLD_RESULTS       ARRAY [0]
1     PIN_FLD_POID          POID [0] NULL
1 PIN_FLD_LOGIN           STR [0] ""

result:

XOP PCM_OP_GLOBAL_SEARCH 0 1
XOP: opcode 25, flags 0
# number of field entries allocated 3, used 3
0 PIN_FLD_POID           POID [0] 0.0.0.1 /search -1 0
0 PIN_FLD_RESULTS       ARRAY [0] allocated 2, used 2
1     PIN_FLD_POID          POID [0] 0.0.0.2 /service/pcm_client 1 1
1     PIN_FLD_LOGIN          STR [0] "root.0.0.0.2"
0 PIN_FLD_RESULTS       ARRAY [1] allocated 2, used 2
1     PIN_FLD_POID          POID [0] 0.0.0.1 /service/pcm_client 1 1
1     PIN_FLD_LOGIN            STR [0] "root.0.0.0.1"
```

Bill Numbering

Applications must avoid hard-coding bill numbers. Bill numbers are coded to the database schema they were created in, and BRM relies on the numbering scheme. The **/data/sequence** storable class tracks bill numbers to ensure that they are unique. This storable class makes sure that bill numbers are unique across database schemas.

41

Creating BRM Client Applications by Using the MTA Framework

Learn about the multithreaded application (MTA) framework in Oracle Communications Billing and Revenue Management (BRM) and how to use it to create BRM multithreaded client applications.

Topics in this document:

- About the BRM MTA Framework
- Using the BRM MTA Framework
- Creating a Multithreaded BRM Client Application
- Customizing BRM Multithreaded Client Applications
- Configuring your Multithreaded Application
- Using Multithreaded Applications with Multiple Database Schemas
- MTA Policy Opcode Hooks
- MTA Callback and Helper Functions

See also:

- Using Transactions in Your Client Application
- Creating Applications that Run on Multischema Systems

About the BRM MTA Framework

You use BRM multithreaded application (MTA) framework to create customizable multithreaded BRM client applications. A multithreaded application uses multiple threads that run in parallel to process a single task. By using multiple worker threads, BRM MTAs are able to process jobs more quickly.

BRM MTAs use a standard program structure, making it easier to code and maintain. In each application, one main thread is responsible for getting data from the BRM database, and a number of worker threads process the job in parallel. The BRM MTA framework manages all thread handling seamlessly, allowing your application to ignore thread management.

Typically, you use a multithreaded BRM client application when you have a *large* job that can be grouped into batches and processed concurrently. For example, BRM's **pin_deferred_act**, **pin_bill_accts**, **pin_collect**, and **pin_cycle_fees** billing utilities use the MTA framework to retrieve information from the BRM database for the accounts that they process.

The BRM MTA framework is based on a multi-layered architecture that allows you to create customizable BRM MTAs. For information about this architecture, see "BRM MTA Framework Layers".

The BRM MTA framework provides function and opcode hooks that you implement to create customizable multithreaded applications. Each callback function and policy opcode is called at fixed places during application execution. For information about these execution stages, see "MTA Stages".

Information about the application, such as configuration settings and search flists are stored in a global flist. The global flist makes application information available to all three layers: Framework, Application, and Customization. For information about the global flist structure, see "MTA Global Flist Structure".

Function hooks are provided as MTA callback functions, which you use to implement your application's business logic in the Application layer. See "Creating a Multithreaded BRM Client Application".

Opcode hooks are custom policy opcodes that you write to customize the business logic in the Customization layer. See "Customizing BRM Multithreaded Client Applications".

Each callback function and policy opcode provides a specific functionality. For details, see "Using the BRM MTA Framework".

BRM MTA Framework Layers

The BRM MTA framework has three layers:

- **Framework layer**

  This layer is implemented at the application tier in the BRM four-tier architecture. This layer implements the main thread that controls application workflow. The main thread performs database searches, distributes jobs to worker threads, and calls the callback functions in the Application layer and the custom policy opcodes in the Customization layer.

- **Application layer**

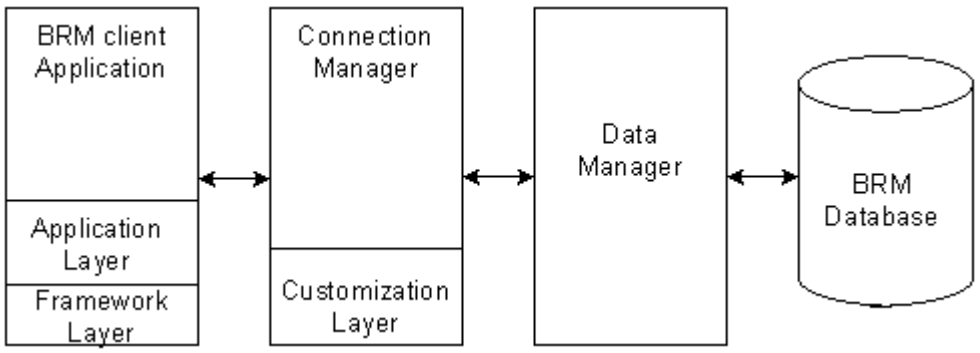
  This layer is implemented at the application tier. This layer consists of the MTA callback functions. You use callback functions to implement your application business logic, such as calling billing opcodes to perform billing or generate invoices.

- **Customization layer**

  This layer is implemented at the Connection Manager (CM) tier. This layer consists of custom policy opcodes. You use policy opcodes to customize the application business logic implemented in the Application layer.

Figure 41-1 shows the architecture of the BRM MTA framework layers:

Figure 41-1 BRM MTA Framework Architecture

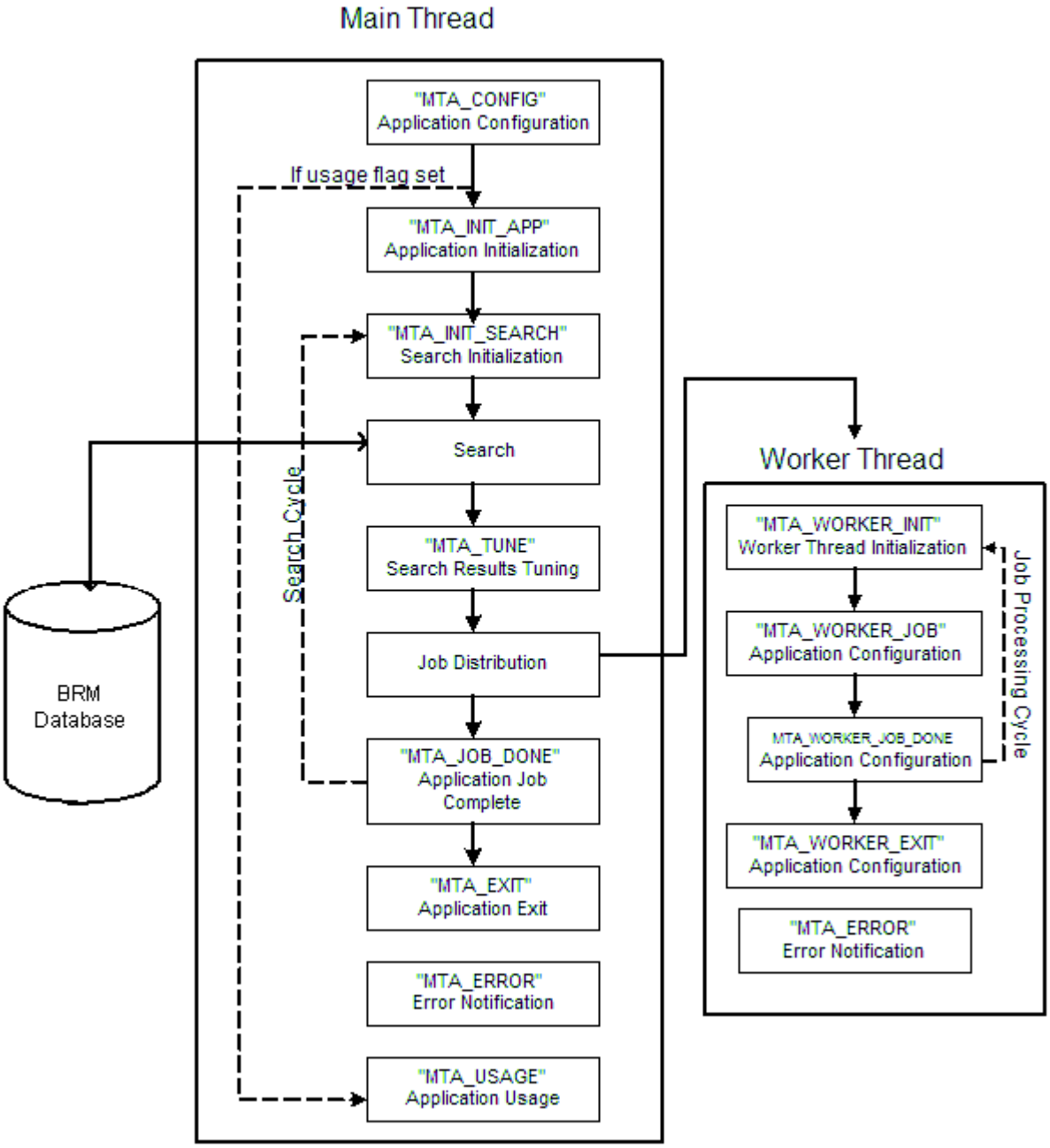


MTA Stages

Each BRM multithreaded application has a standard program structure and standard execution stages. The main thread manages the application workflow by calling the MTA functions and policy opcodes in a set order at each execution stage.

Figure 41-2 shows the BRM MTA execution stages and workflow.

Figure 41-2 BRM MTA Execution Stages and Workflow



MTA_CONFIG Execution Stage

Each BRM multithreaded client application requires a configuration file (**pin.conf**) and can have command-line parameters. The MTA framework performs default application configuration based on the information in the **pin.conf** file and command-line parameters. You can write custom policy opcodes to provide custom configurations.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_config
2. MTA_CONFIG
3. pin_mta_post_config

MTA_INIT_APP Execution Stage

Application initialization occurs after the application has been configured successfully. The main thread spawns a number of worker threads, and the application completes all tasks that are required before the main search execution is performed.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_init_app
2. MTA_INIT_APP
3. pin_mta_post_init_app

MTA_INIT_SEARCH Execution Stage

During search initialization, the search flist is prepared. This flist is provided as the input to the search opcodes. You can write custom policy opcodes to modify the search flist.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_init_search
2. MTA_INIT_SEARCH
3. pin_mta_post_init_search

Search Execution

At the search execution stage, the main thread calls the search opcodes (PCM_OP_SEARCH, PCM_OP_STEP_SEARCH, and PCM_OP_STEP_NEXT) to find the objects or search results.

> **Note:**
>
> The search results are sent to the client a block at a time. The block size is equal to the fetch size specified in the application's **pin.conf** file. The search opcodes are called as many times as needed until all the search results are received. For more information, see "Configuring your Multithreaded Application".

MTA_TUNE Execution Stage

After a block of search results is passed to the application, the results can be tuned or modified before they are distributed in batches to the worker threads for processing.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_tune
2. MTA_TUNE
3. pin_mta_post_tune

Job Distribution

The basic purpose of a BRM multithreaded client application is to manage the distribution of a job to worker threads. After search execution and search results tuning, the main thread adds the search results to the job pool and notifies worker threads to begin processing.

MTA_JOB_DONE Execution Stage

After worker threads have been notified of an available job in the job pool, the main thread waits for notification back from the worker threads that their assigned jobs are completed. When all results from the main search have been processed and the job pool is empty, the next search cycle is run until there are no more search results remaining in the database.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_job_done
2. MTA_JOB_DONE
3. pin_mta_post_job_done

MTA_EXIT Execution Stage

When there are no more jobs to process, the application terminates all threads, closes the database connection, and exits.

The main thread calls the following MTA callback functions and the custom policy opcode hook in this order:

1. pin_mta_exit
2. MTA_EXIT
3. pin_mta_post_exit

MTA_WORKER_INIT Execution Stage

Worker threads are spawned at the application initialization stage. After initialization, the worker threads remain in wait mode until they are notified by the main thread of an available job in the job pool.

The following MTA callback functions and the policy opcode hook are called in this order by each worker thread:

1. pin_mta_worker_init

2. MTA_WORKER_INIT
3. pin_mta_post_worker_init

MTA_WORKER_JOB Execution Stage

After worker threads have been notified of an available job in the job pool, they receive their assigned batch of search results and then call the main opcode to process the work. The input flist for the main opcode is prepared here prior to the opcode call.

The following MTA callback functions and the policy opcode hook are called by each worker thread in this order:

1. pin_mta_worker_job
2. MTA_WORKER_JOB
3. pin_mta_post_worker_job

Worker Thread Job Execution

Each worker thread calls the "pin_mta_worker_opcode" callback function and passes the search results for processing.

MTA_WORKER_JOB_DONE Execution Stage

When worker threads have completed their assigned job, they notify the main thread and return to wait mode until more work becomes available.

After a worker thread has completed its job, it calls the following callback functions and policy opcode hook to perform any required tasks:

1. pin_mta_worker_job_done
2. MTA_WORKER_JOB_DONE
3. pin_mta_post_worker_job_done

MTA_WORKER_EXIT Execution Stage

Worker threads are terminated when there are no more jobs for the application to process.

The following MTA callback functions and the custom policy opcode hook are called in this order by each worker thread to perform any tasks before the thread exits:

1. pin_mta_worker_exit
2. MTA_WORKER_EXIT
3. pin_mta_post_worker_exit

MTA Global Flist Structure

The BRM MTA framework includes a global flist that stores information specific to the application, such as configuration settings, search flists, and search results. Information stored in the global flist is accessed by the MTA callback functions and custom policy opcodes.

The global flist contains the following fields:

```
0 PIN_FLD_CONFIG_OBJ           SUBSTRUCT [0]
0 PIN_FLD_APPLICATION_INFO     SUBSTRUCT [0]
```

```
0 PIN_FLD_SEARCH_FLIST          SUBSTRUCT [0]
0 PIN_FLD_SEARCH_RESULTS        SUBSTRUCT [0]
0 PIN_FLD_EXTENDED_INFO         SUBSTRUCT [0]
0 PIN_FLD_OPERATION_INFO        SUBSTRUCT [0]
```

The PIN_FLD_CONFIG_OBJ substruct is populated with information from the **/config/mta** object. This information is used by the BRM MTA framework to determine which custom policy opcodes to call during application execution. For more information about the fields in the PIN_FLD_CONFIG_OBJ substruct, see "Configuring the MTA Policy Opcodes".

The PIN_FLD_APPLICATION_INFO substruct contains the application's configuration settings.

This substruct includes the following fields:

```
0 PIN_FLD_APPLICATION_INFO SUBSTRUCT [0]
1    PIN_FLD_NAME               STR [0]
1    PIN_FLD_CHILDREN           INT [0]
1    PIN_FLD_STEP_SIZE          INT [0]
1    PIN_FLD_BATCH_SIZE         INT [0]
1    PIN_FLD_FETCH_SIZE         INT [0]
1    PIN_FLD_NUM_RETRIES        INT [0]
1    PIN_FLD_FLAGS              INT [0]
1    PIN_FLD_MAX_ERROR          INT [0]
1    PIN_FLD_MAX_TIME           INT [0]
1    PIN_FLD_HOTLIST_FILENAME   STR [0]
1    PIN_FLD_MONITOR_FILENAME   STR [0]
1    PIN_FLD_LOGFILE            STR [0]
1    PIN_FLD_LOGLEVEL           INT [0]
1    PIN_FLD_POID_VAL           POID [0]/*Specifies the DB no.
```

where:

- PIN_FLD_NAME is the application name.
- PIN_FLD_FLAGS contains application bit flags. Flags can be added by Application layer and Customization layer developers. See **pin_mta.h** for the default MTA flags.

For details about all other PIN_FLD_APPLICATION_INFO substruct fields, see "Configuring your Multithreaded Application".

The PIN_FLD_SEARCH_FLIST substruct contains the search flist that is passed to the search opcodes.

This substruct includes the following fields:

```
0 PIN_FLD_SEARCH_FLIST          SUBSTRUCT [0]
1    PIN_FLD_POID             POID [0]
1     PIN_FLD_FLAGS            INT [0]
1    PIN_FLD_TEMPLATE         STR [0]
1    PIN_FLD_ARGS             ARRAY [1]
2        PIN_FLD_XXX              STR [0]
1    PIN_FLD_RESULTS          ARRAY [0]
2        PIN_FLD_XXX              POID [0]
1    PIN_FLD_FILENAME         STR [0]
1    PIN_FLD_COUNT             INT [0]
```

For details about these fields, see "Configuring your Multithreaded Application".

The PIN_FLD_SEARCH_RESULTS substruct contains the search flist that is passed to the search opcodes.

This substruct includes the following fields:

```
0 PIN_FLD_SEARCH_RESULTS   ARRAY [0]
1    PIN_FLD_MULTI_RESULTS    ARRAY [0]
2        PIN_FLD_RESULTS          ARRAY [0]
3            PIN_FLD_XXX              POID [0]
```

where:

- PIN_FLD_MULTI_RESULTS is an array containing the search results. The number of results is equal to the step size specified in the **pin.conf** file.
- PIN_FLD_RESULTS is an array that specifies the objects received from the database.

The PIN_FLD_EXTENDED_INFO substruct is reserved for the Customization layer.

The PIN_FLD_OPERATION_INFO substruct contains statistics and audit-related information.

This substruct includes the following fields:

```
1    PIN_FLD_PID             INT [0]
1    PIN_FLD_HOSTNAME        STR [0]
1    PIN_FLD_START_T         TSTAMP [0]
1    PIN_FLD_THREAD_INFO     ARRAY [0]
2          PIN_FLD_START_T         TSTAMP [0]
2         PIN_FLD_END_T           TSTAMP [0]
2         PIN_FLD_ERROR_INFO      ARRAY [0]
3            PIN_FLD_ERROR_NUM       INT [0]
3            PIN_FLD_SYS_ERROR_NUM   INT [0]
3            PIN_FLD_ERROR_CODE      STR [0]
3            PIN_FLD_ERROR_DESCR     STR [0]
3             PIN_FLD_TRACKING_ID     STR [0]
```

where:

- PIN_FLD_PID specifies the process ID number.
- PIN_FLD_HOSTNAME specifies the host where the application runs.
- PIN_FLD_START_T specifies the process or main thread start time.
- PIN_FLD_END_T specifies the process end time.
- PIN_FLD_ERROR_NUM specifies the total number of errors for all threads.
- PIN_FLD_SYSTEM_ERROR_NUM specifies the total number of system errors.
- PIN_FLD_ERROR_CODE specifies the error code from the application **pinlog** file.
- PIN_FLD_ERROR_DESCRIPTION is the description of the error from the application **pinlog** file.
- PIN_FLD_TRACKING_ID specifies the correlation ID from the application **pinlog** file.

Using the BRM MTA Framework

The BRM MTA framework is compiled as a static library. It provides a set of callback functions as hooks that you implement to develop multithreaded client applications for BRM.

The MTA framework is included in the BRM SDK. For installation instructions and an overview, see "About BRM SDK".

The MTA framework includes the files listed in Table 41-1, in the *BRM_SDK_home* directory:

Table 41-1 Files Included in MTA Framework

File	Description
/include/pin_mta.h	MTA header file
/lib/libmta.a	MTA library file
/bin/pin_mta_monitor	Sample monitoring utility
/source/apps/mta_sample/pin_mta_test.c	Sample application using the MTA framework
/source/apps/mta_sample/Makefile	Makefile to build the sample application, **pin_mta_test.c**
/source/apps/mta_sample/pin.conf	Sample configuration file

MTA Callback Functions

The BRM MTA callback functions (see Table 41-2) are hooks. You implement the functions in your application by providing application-specific contents.

> **Note:**
>
> You do not have to implement all of the callback functions; however, you *must* implement **pin_mta_config** to process application command-line parameters, **pin_mta_init_search** to specify search criteria, and **pin_mta_worker_opcode** to specify the main opcode call. You implement other functions as needed.

Table 41-2 MTA Callback Functions

Function	Description
pin_mta_config	Can be used for default application configuration.
pin_mta_exit	Can be used for any tasks that are required before the application exits.
pin_mta_init_app	Can be used for application initialization and all required tasks before execution of main search.
pin_mta_init_search	Can be used for preparation of the search flist.
pin_mta_job_done	Can be used for any tasks that are required after search results have been processed, such as validation and logging.
pin_mta_post_config	Can be used for post-configuration tasks such as configuration validation and logging.
pin_mta_post_exit	Can be used for any tasks that are required before the application exits, such as validation and logging.
pin_mta_post_init_app	Can be used for any post-initialization tasks such as initialization validation and logging.
pin_mta_post_init_search	Can be used for any tasks that are required after search initialization, such as validation and logging.
pin_mta_post_job_done	Can be used for any tasks that are required after search results have been processed, such as validation and logging.

Table 41-2 (Cont.) MTA Callback Functions

Function	Description
pin_mta_post_tune	Can be used for any tasks that are required after search results have been tuned, such as validation and logging.
pin_mta_post_usage	Can be used to display application help information.
pin_mta_post_worker_exit	Performs any cleanup tasks and logging after a worker thread exits.
pin_mta_post_worker_init	Can be used for any tasks that are required after worker threads are initialized, such as validation and logging.
pin_mta_post_worker_job	Can be used for any validation of the search results received by worker threads and any logging that is required.
pin_mta_post_worker_job_done	Performs any cleanup tasks and logging that are required after a worker thread completes its assigned job.
pin_mta_tune	Can be used for any filtration and tuning of the search results before the results are distributed to worker threads.
pin_mta_usage	Can be used to prepare application help information for display.
pin_mta_worker_exit	Performs cleanup tasks and other functions required before a worker thread exits.
pin_mta_worker_init	Can be used for initialization of the worker thread.
pin_mta_worker_job	Can be used to prepare the main opcode input flist.
pin_mta_worker_job_done	Can be used for any functions that are required after a worker thread completes its assigned job.
pin_mta_worker_opcode	Runs the main opcode to process the job.

MTA Helper Functions

Table 41-3 lists the MTA helper functions used to manipulate data in global flist data structures.

Table 41-3 MTA Helper Functions

Function	Description
pin_mta_get_decimal_from_pinconf	Loads decimal fields from the **pin.conf** file.
pin_mta_get_int_from_pinconf	Loads integer fields from the **pin.conf** file.
pin_mta_get_str_from_pinconf	Loads string fields from the **pin.conf** file.
pin_mta_global_flist_node_get_no_lock	Gets the global flist field; does not set the lock on the field; multithread unsafe.
pin_mta_global_flist_node_get_with_lock	Gets the global flist field; sets the lock on the field to prevent access; multithread safe.
pin_mta_global_flist_node_put	Puts the global flist field to the flist specified; multithread safe.
pin_mta_global_flist_node_release	Releases the lock from the global flist field; multithread safe.
pin_mta_global_flist_node_set	Sets the global flist field to the flist specified; multithread safe.
pin_mta_main_thread_pcm_context_get	Gets the main thread context.

MTA Policy Opcode Hooks

This section lists MTA policy opcode hooks.

All MTA policy opcodes use Transition: supports.

For information about the flist specification, see *Opcode Flist Reference*.

To customize your BRM multithreaded client application, you implement the MTA policy opcode hooks (listed in Table 41-4) in your application by providing application-specific contents.

> **Note:**
>
> You do not have to implement all of the policy opcodes. You specify your own policy opcode names. For example, for MTA_CONFIG policy opcode, you might use PCM_OP_MTA_POL_CONFIG.

Table 41-4 MTA Policy Opcode Hooks

Policy opcode	Description
MTA_CONFIG	Called at MTA_CONFIG execution stage. Allows customization of the default application configuration.
MTA_ERROR	Allows processing of error notifications.
MTA_EXIT	Called at MTA_EXIT execution stage. Allows any processing that may be required before the application exits.
MTA_INIT_APP	Called at MTA_INIT_APP execution stage. Allows customization of the default application initialization.
MTA_INIT_SEARCH	Called at MTA_INIT_SEARCH execution stage. Allows customization of the search flist.
MTA_JOB_DONE	Called at MTA_JOB_DONE execution stage. Allows any processing that may be required at the completion of search results processing.
MTA_TUNE	Called at MTA_TUNE execution stage. Allows modification of the search results.
MTA_USAGE	Allows display of application help information.
MTA_WORKER_EXIT	Called at MTA_WORKER_EXIT execution stage. Allows any processing that may be required before the worker thread exits.
MTA_WORKER_INIT	Called at MTA_WORKER_INIT execution stage. Allows customization of worker thread initialization.
MTA_WORKER_JOB	Called at MTA_WORKER_JOB execution stage. Allows modification of search results received by the worker thread and preparation of the input flist passed to the main opcode.
MTA_WORKER_JOB_DONE	Called at MTA_WORKER_JOB_DONE execution stage. Allows any processing that may be required at completion of the worker thread job assignment.

Creating a Multithreaded BRM Client Application

Before you create a multithreaded BRM client application, you should have a good understanding of the BRM MTA execution flow and the global flist structure. See "About the BRM MTA Framework".

You create a multithreaded BRM client application by implementing the MTA callback functions in your application and by providing application-specific content.

These callback functions are called by the MTA framework at fixed execution points to configure, initialize, search, process data, and exit the application. For information about the syntax and description of these functions, see "MTA Callback Functions".

The BRM MTA framework supports three search options; see "Searching Different Data Sources".

For information about displaying usage information, see "Displaying Application Help Information".

For processing errors that occur during application execution, see "Error Notifications".

The sample file *BRM_SDK_home***/source/samples/apps/c/mta_sample/pin_mta_test.c** provides sample implementations of MTA callback functions. Use this as a code sample to build your own multithreaded application. For information on how to compile and run the sample application, see "About Using the PCM C Sample Programs" in *BRM Developer's Reference*.

For general information on creating new client applications, see "Adding New Client Applications".

Each BRM client application has its own configuration file (**pin.conf**). After you build your application, you need to create a configuration file. See "Configuring your Multithreaded Application".

To customize your multithreaded application, you implement the MTA policy opcode hooks. See "Customizing BRM Multithreaded Client Applications".

Searching Different Data Sources

The BRM MTA framework supports three search options: search for objects stored in the BRM database, search for objects stored in a file, and passing objects directly to the search opcode. You specify the search option in the PIN_FLD_SEARCH_FLIST substruct in the global flist.

To search for objects in the BRM database, you specify the POID of the **/search** object that defines the search template.

For example:

```
0 PIN_FLD_SEARCH_FLIST     SUBSTRUCT [0]
1     PIN_FLD_POID             POID [0] "/search/pin" -1 0
1     PIN_FLD_FLAGS           INT [0] 0
1     PIN_FLD_TEMPLATE          STR [0] "select X from /account where F1 = V1"
1     PIN_FLD_ARGS             ARRAY [1]
2         PIN_FLD_FIRST_NAME        STR [0] "name"
1     PIN_FLD_RESULTS           ARRAY [0]
2         PIN_FLD_POID              POID [0]
```

To search for objects in a file, you specify the name of the file where the objects are stored.

For example:

```
0 PIN_FLD_SEARCH_FLIST    SUBSTRUCT [0]
1    PIN_FLD_FILENAME          STR [0] "file name"
1    PIN_FLD_COUNT             INT [0] 2
```

> **Note:**
>
> The file specified in PIN_FLD_FILENAME *must* be a text file and *must* have the same format as the search results flist.

The following is an example of the text file:

```
0 PIN_FLD_RESULTS          ARRAY [0]
1    PIN_FLD_ACCOUNT_OBJ       POID [0] "/account" 123 0
0 PIN_FLD_RESULTS          ARRAY [1]
1    PIN_FLD_ACCOUNT_OBJ       POID [0] "/account" 345 0
```

To pass objects directly in the search flist, you specify the POIDs of the objects.

For example:

```
0 PIN_FLD_POID              POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_RESULTS          ARRAY [0]
1    PIN_FLD_ACCOUNT_OBJ       POID [0] "/account" 123 0
0 PIN_FLD_RESULTS          ARRAY [1]
1    PIN_FLD_ACCOUNT_OBJ       POID [0] "/account" 345 0
```

> **Note:**
>
> When objects are provided in the search template, the data is taken from PIN_FLD_SEARCH_FLIST and put into PIN_FLD_SEARCH_RESULTS. Do not free memory from PIN_FLD_SEARCH_FLIST, otherwise the application will error out. If your application performs the search operation in a loop, add new data to PIN_FLD_SEARCH_FLIST node by preparing the flist in the **pin_mta_init_search** function. The application will exit as soon as no new data is provided in PIN_FLD_SEARCH_FLIST node.

For information about the search flist and search results flist, see "MTA Global Flist Structure".

Displaying Application Help Information

The BRM MTA framework provides callback functions that you can implement in your application to display application help information. The usage callback function is called by specifying the **-help** parameter in the command line or when the application fails to configure or initialize because the command-line options specified were invalid or incomplete. When an error occurs during application configuration, the MTA framework sets the PIN_FLD_FLAGS field in the PIN_FLD_APPLICATION_INFO substruct in the global flist. For a list of all predefined flags, see **pin_mta.h.**

> **Note:**
>
> The command-line parameters **-help**, **-verbose**, and **-test** are processed by the MTA framework layer. When these parameters are specified with additional parameters (for example, **-verbose** *xyz*), the framework layer processes the **-verbose** parameter and ignores *xyz*. In this case, you might want to display a usage error message in your custom application. To do this, set the MTA flag in the **pin_mta_config** callback function as follows:
>
> **mta_flags = mta_flags | MTA_FLAG_VERSION_NEW**
>
> When this flag is set, the MTA framework generates a usage error when the **-help**, **-verbose**, or **-test** parameter is specified with additional parameters.

To display application help information, the main thread calls the following callback functions and the policy opcode hook in this order for the **-help**, **-verbose**, or **-test** parameters:

1. pin_mta_usage
2. MTA_USAGE
3. pin_mta_post_usage

You implement **pin_mta_usage** to prepare the help text message, and you implement **pin_mta_post_usage** to display the message. You can customize the help text by implementing the usage policy opcode hook.

> **Note:**
>
> You *must* set the PIN_FLD_DESCR field in the PIN_FLD_EXTENDED_INFO substruct in the global flist to the help text, so that it can be accessed by the usage policy opcode and **pin_mta_post_usage.**

Depending on the parameter specified in the command line, the framework layer processes the parameters as follows:

- **-help**: Sets the PIN_FLD_FLAGS field in the PIN_FLD_APPLICATION_INFO substruct in the global flist to MTA_FLAG_USAGE_MSG, calls **pin_mta_usage** and then **pin_mta_exit**, and displays the valid command line options on the standard output.
- **-verbose**: Sets the PIN_FLD_FLAGS field in the PIN_FLD_APPLICATION_INFO substruct in the global flist to MTA_FLAG_VERBOSE_MODE; flushes the **stdout** buffer; and displays the number of data errors encountered, the total number of errors encountered, and the additional output for threads information on the standard output.
- **-test**: Sets the PIN_FLD_FLAGS field in the PIN_FLD_APPLICATION_INFO substruct in the global flist to MTA_FLAG_TEST_MODE, and displays the total number of prepared job units for batch processing and the total number of spawned worker threads on the standard output.

For information about the global flist, see "MTA Global Flist Structure".

The application usage policy opcode is called *only* if it's configured in **/config/mta**. See "Configuring the MTA Policy Opcodes".

Error Notifications

In a multithreaded application, errors can occur in the main thread or the worker threads. In BRM MTA, errors that occur in the main thread are handled differently than errors in the worker threads.

Generally, an error in the main thread is an indication of a serious problem that prevents the application from continuing its normal execution. When an error occurs in the main thread, the BRM MTA framework exits the application immediately.

Errors that occur in the worker threads are not as severe and therefore it is not necessary for the application to exit immediately. When an error occurs in a worker thread, the MTA framework checks to see if the maximum error threshold has been reached. The MTA framework exits the application when the number of errors in the worker threads exceeds the threshold.

When the application exits due to error conditions, the MTA framework calls the error notification policy opcode to allow exit processing that may be required at the Customization layer.

> **Note:**
>
> The error notification policy opcode is called by the BRM MTA framework only if it is configured in **/config/mta**. See "Configuring the MTA Policy Opcodes".

You can implement the error notification policy opcode in your application to process errors. For information about the input and output flist, see "MTA_ERROR".

Customizing BRM Multithreaded Client Applications

Before you customize a multithreaded BRM client application, you should have a good understanding of the BRM MTA execution flow and the global flist structure. See "About the BRM MTA Framework".

The BRM MTA framework provides policy opcode hooks for customization of BRM MTAs. For example, you can create a custom billing utility by customizing BRM's **pin_bill_accts** MTA. You implement these policy opcode hooks by providing your business-specific contents.

For more information on how to define and configure a new opcode, see Defining New Opcodes. The **pin.conf** entry for **ops_fields_extension_file** must be defined to permit the MTA applications to recognize custom opcodes.

These policy opcode hooks are called by the BRM MTA framework at fixed places during the application execution. Unlike the MTA callback functions, the MTA policy opcode hooks *must* be configured using the **/config/mta** object.

To customize a BRM MTA, you need to do the following:

1. About Shadow Objects
2. Configuring the MTA Policy Opcodes

Implementing the MTA Policy Opcodes

You use the MTA policy opcode hooks to customize multithreaded BRM client applications. They do not have default implementations; therefore, they aren't in the BRM System Facilities Modules (FM). You need to write a *custom* policy FM to include your custom policy opcodes.

The policy opcode hooks do not have predefined opcode names; you create your own. For example, you could name the MTA_CONFIG policy opcode PCM_OP_MTA_POL_CONFIG.

For more information about writing a custom FM, see "Writing a Custom Facilities Module".

> **Note:**
>
> The MTA policy opcode hooks have predefined input and output specifications. You *must* write your custom policy opcodes based on these specifications.

For a list of all the policy opcodes and details about the input and output specifications, see "MTA Policy Opcode Hooks".

Configuring the MTA Policy Opcodes

The MTA policy opcodes are called by the main thread at specific execution stages in the application workflow. At application startup, the MTA framework reads the **/config/mta** object to determine if custom policy opcodes are implemented at the Customization layer.

You use **testnap** or Developer Center to populate the PIN_FLD_OPCODE_MAP array in the **/config/mta** object to specify the policy opcode names and the execution stages that these opcodes are called from.

For information about the fields and field types in **/config/mta**, see **/config/mta.**

> **Note:**
>
> The execution stage names are predefined. You must use these names when populating the PIN_FLD_FUNCTION field in the **/config/mta** object. Otherwise, the application will not load the object at startup and it will exit instead. For details about the policy opcode and its execution stage names, see "MTA Policy Opcode Hooks".

In this example, the MTA application **pin_mta_test** calls custom policy opcodes at the MTA_CONFIG, MTA_INIT_ERROR, and MTA_USAGE execution stages:

```
0 PIN_FLD_CONFIG_MTA    ARRAY [0]
1    PIN_FLD_NAME            STR [0] "pin_mta_test"
1    PIN_FLD_OPCODE_MAP    ARRAY [0]
2        PIN_FLD_FUNCTION        STR [0] "MTA_CONFIG"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_CONFIG"
1    PIN_FLD_OPCODE_MAP    ARRAY [1]
2        PIN_FLD_FUNCTION        STR [0] "MTA_ERROR"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_ERROR"
1    PIN_FLD_OPCODE_MAP    ARRAY [2]
2        PIN_FLD_FUNCTION        STR [0] "MTA_USAGE"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_USAGE"
```

In this example, the MTA application **pin_mta_test** calls custom policy opcodes at the MTA_INIT_SEARCH, MTA_TUNE, and MTA_WORKER_JOB_DONE execution stages:

```
0 PIN_FLD_CONFIG_MTA    ARRAY [0]
1    PIN_FLD_NAME            STR [0] "pin_mta_test"
1    PIN_FLD_OPCODE_MAP    ARRAY [0]
2        PIN_FLD_FUNCTION        STR [0] "MTA_INIT_SEARCH"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_INIT_SEARCH"
1    PIN_FLD_OPCODE_MAP    ARRAY [1]
2        PIN_FLD_FUNCTION        STR [0] "MTA_TUNE"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_TUNE"
1    PIN_FLD_OPCODE_MAP    ARRAY [2]
2        PIN_FLD_FUNCTION        STR [0] "MTA_WORKER_JOB_DONE"
2        PIN_FLD_NAME            STR [0] "PCM_OP_MTA_POL_WORKER_JOB_DONE"
```

In extremely rare cases, you might create a dynamic library to implement custom MTA callback functions that is not feasible by using the MTA policy opcodes.

You can use the **/config/mta** object to specify custom callback functions. In this case, you must populate the PIN_FLD_FUNCTION_MAP array to map the custom function name to the MTA callback function name. The function names specified in PIN_FLD_FUNCTION must match the default MTA callback function names. The MTA framework will call the custom callback function instead of the default callback function. For a list of all MTA callback functions, see "MTA Callback Functions".

> **Note:**
>
> If custom functions are configured, the MTA framework calls the custom function *in place of* the default MTA callback function. In this case, the default functionality is lost.

```
0 PIN_FLD_CONFIG_MTA    ARRAY [0]
1    PIN_FLD_LIBRARY         STR [0] "libpin_mta_test_lib.so"
1    PIN_FLD_NAME            STR [0] "pin_mta_test"
1    PIN_FLD_FUNCTION_MAP  ARRAY [0]
2        PIN_FLD_FUNCTION        STR [0] "pin_mta_post_init_app"
2        PIN_FLD_NAME            STR [0] "pin_mta_post_init_app_lib"
1    PIN_FLD_FUNCTION_MAP  ARRAY [1]
2        PIN_FLD_FUNCTION        STR [0] "pin_mta_post_tune"
2        PIN_FLD_NAME            STR [0] "pin_mta_post_tune_lib"
1    PIN_FLD_FUNCTION_MAP  ARRAY [2]
2        PIN_FLD_FUNCTION        STR [0] "pin_mta_config"
2        PIN_FLD_NAME            STR [0] "pin_mta_config_lib"
1    PIN_FLD_FUNCTION_MAP  ARRAY [3]
2        PIN_FLD_FUNCTION        STR [0] "pin_mta_init_search"
2        PIN_FLD_NAME            STR [0] "pin_mta_init_search_lib"
1    PIN_FLD_FUNCTION_MAP  ARRAY [4]
2        PIN_FLD_FUNCTION        STR [0] "pin_mta_tune"
2        PIN_FLD_NAME            STR [0] "pin_mta_tune_lib"
```

Configuring your Multithreaded Application

The sample configuration file in the *BRM_home***/source/samples/apps/C/mta_sample/pin.conf** directory specifies the configuration information for a multithreaded application to connect to the BRM database and process data. (*BRM_home* is the directory in which the BRM server software is installed.)

For each multithreaded application that you create, you need to include a similar configuration file in your application's directory with entries specific to the application.

For a list of configuration file entries, see "Creating Configuration Files for BRM Utilities" in *BRM System Administrator's Guide*.

> **Note:**
>
> You can create additional configuration entries that are required for your MTA application.

Applying Configuration Entries to Specific Utilities

To apply an entry to all the MTA applications, use **pin_mta**. For example:

- pin_mta children 5
- pin_mta fetch_size 10000
- pin_mta per_batch 100

To apply an entry only to a particular MTA, use **pin**_*application_name*. For example, **pin_bill_accts**:

- pin_bill_accts children 5
- pin_bill_accts fetch_size 10000
- pin_bill_accts per_batch 100

Using Multithreaded Applications with Multiple Database Schemas

To use a multithreaded application with a multischema BRM installation, you must change the **multi_db** entry in the application **pin.conf** file. For example, to use the global search feature to search across schemas, you must enable multischema support. Set the **multi_db** entry to **1** to enable multischema support.

MTA Policy Opcode Hooks

This section lists MTA policy opcode hooks.

MTA_CONFIG

This policy opcode allows customization of the default application configuration.

This policy opcode is called by the MTA framework at the MTA_CONFIG execution stage. It is called after **pin_mta_config**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to perform custom application configuration. For example, you can write code to retrieve configuration parameters from a database object.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta** object. See "Configuring the MTA Policy Opcodes".

MTA_ERROR

This policy opcode allows processing of error notifications.

This policy opcode is called by the MTA framework when an error occurs in the main thread or when the maximum error threshold has been reached for the worker threads. You set the maximum error limit in the application's **pin.conf** file. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to process error notifications at the Customization layer.

For example, you can write code to log appropriate error messages in the application's log file.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_EXIT

This policy opcode allows any processing required before the application exits.

This policy opcode is called by the MTA framework at the MTA_EXIT execution stage. This is when there are no more jobs to be processed. This policy opcode can also be called when an error occurs during application execution. This policy opcode is called after **pin_mta_exit**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to process errors, perform logging, clean up procedures, or other functionality.

For example, you can write code to process errors and log appropriate error messages in the application's log file.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_INIT_APP

This policy opcode allows customization of the default application initialization.

This policy opcode is called by the MTA framework at the MTA_INIT_APP execution stage. It is called after **pin_mta_init**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to perform custom application initialization.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_INIT_SEARCH

This policy opcode enables customization of the search flist.

This policy opcode is called by the MTA framework at the MTA_INIT_SEARCH execution stage. It is called after **pin_mta_init_search**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to modify the search flist that is passed to the search opcodes PCM_OP_SEARCH, PCM_OP_STEP_SEARCH, and PCM_OP_STEP_NEXT. In a multischema search, the search flist is passed to PCM_OP_GLOBAL_SEARCH, PCM_OP_GLOBAL_STEP_SEARCH, and PCM_OP_GLOBAL_STEP_NEXT.

For example, you can use this opcode to customize information in the search template.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_JOB_DONE

This policy opcode allows any processing required at completion of search results processing.

This policy opcode is called by the MTA framework at the MTA_JOB_DONE execution stage. This is after the worker threads have completed their assigned jobs, there are no more search results to process, and the job pool is empty. This policy opcode is called after **pin_mta_job_done**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to perform validation, logging, or other functionality that may be required.

For example, you can write code to analyze the percentage of a job that was processed successfully and to create a log of those threads that failed.

If another search is required, you can write code to loop through the search cycle again.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_TUNE

This policy opcode allows modification of the search results.

This policy opcode is called by the MTA framework at the MTA_TUNE execution stage. This opcode is called after **pin_mta_tune**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to preprocess the search results before they are distributed to the worker threads for processing.

For example, you can write code to perform validation of the search results and to modify or filter the results.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_USAGE

This policy opcode allows display of application help information.

This policy opcode is called when the user explicitly requests the application's usage information by specifying the **-help** parameter at the command line. This policy opcode is also called by the MTA framework during application configuration when it fails to configure the application using the command-line parameters specified. This policy opcode is called after **pin_mta_usage**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to customize help information.

For example, you can write code to display a custom help message for custom command-line options.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_WORKER_EXIT

This policy opcode allows any processing required before the worker thread exits.

This policy opcode is called by the MTA framework at the MTA_WORKER_EXIT execution stage. This is when the worker thread is notified that the application is about to exit. This policy opcode is called after **pin_mta_worker_exit**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to perform any cleanup procedures or other functionality before the worker thread is terminated.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_WORKER_INIT

This policy opcode allows customization of worker thread initialization.

This policy opcode is called by the MTA framework at the MTA_WORKER_INIT execution stage. It is called after **pin_mta_worker_init**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement for customization of worker thread initialization.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta.** See "Configuring the MTA Policy Opcodes".

MTA_WORKER_JOB

This policy opcode allows modification of search results received by worker threads and preparation of the input flist passed to the main opcode.

This policy opcode is called by the MTA framework at the MTA_WORKER_JOB execution stage. This is when the worker thread has received a batch of search results to be processed and the worker thread prepares the input flist that is passed to the main opcode responsible for processing the search results in the batch. This policy opcode is called after **pin_mta_worker_job**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you implement to perform any processing required before the main opcode is run.

For example, you can write code to modify the main opcode input flist or to modify the search results in the batch.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA_WORKER_JOB_DONE

This policy opcode allows processing required after worker thread job completion.

This policy opcode is called by the MTA framework at the MTA_WORKER_JOB_DONE execution stage. This is when the worker thread notifies the main thread that it has completed

processing the batch of search results and is waiting for the next batch. This policy opcode is called after **pin_mta_worker_job_done**. See *BRM Developer's Reference*.

By default, this policy opcode is an empty hook that you can implement to perform any processing that may be required after the worker thread has completed the assigned job.

For example, you can write code to validate or analyze the output flist from the main opcode call.

> **Note:**
>
> This policy opcode is called *only* if it is configured in the **/config/mta**. See "Configuring the MTA Policy Opcodes".

MTA Callback and Helper Functions

The MTA callback and helper functions are listed here, in alphabetical order.

The MTA helper functions can be used to manipulate data in global flist data structures.

> **Note:**
>
> Multithread-safe helper functions can be used with all MTA callback functions. Multithread-unsafe functions cannot be used with **pin_mta_worker** thread functions.

pin_mta_config

This function processes command-line arguments.

This function is called at application configuration for processing application-specific command-line arguments, configuration settings in the application's **pin.conf** file, or objects. It also sets the usage flag when there is an error during configuration, to halt application execution and display a help message.

Syntax

```
void
pin_mta_config(
          pin_flist_t     *param_flistp,
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

param_flistp
A pointer to the flist containing information about the command-line parameters. Information from *param_flistp* is used to populate the PIN_FLD_APPLICATION_INFO substruct in the application's global flist.

> **Tip:**
>
> Removing elements from *param_flistp* as they are processed may help to recognize unexpected command-line options and to set the usage flag, if necessary.

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_exit

This function shuts down the application.

This function is called when the application is about to exit. It is a hook for implementing functions that are required (such as validation) and logging.

Syntax

```
void
pin_mta_exit(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_get_decimal_from_pinconf

This function loads decimal fields from the **pin.conf** file and sets them to the PIN_FLD_APPLICATION_INFO substruct in the global flist.

> **Note:**
>
> This function is not multithread safe and can be used *only* in **pin_mta_config**.

Syntax

```
void
pin_mta_get_decimal_from_pinconf(
          pin_flist_t     *app_info_flistp,
          char            *pinconf_name,
          int32            field,
          int32            optional,
          pin_errbuf_t    *ebufp);
```

Parameters

app_info_flistp
A pointer to the PIN_FLD_APPLICATION_INFO substruct in the global flist.

pinconf_name
A pointer to the **pin.conf** file.

field
The name of the field in the **pin.conf** file.

optional
Specifies whether the field is optional or mandatory. If it is mandatory and does not exist in the **pin.conf** file, the error buffer is set.

ebufp
A pointer to the error buffer.

pin_mta_get_int_from_pinconf

This function loads integer fields from the **pin.conf** file and sets them to the value in the PIN_FLD_APPLICATION_INFO substruct in the global flist.

> **Note:**
>
> This function is multithread unsafe and cannot be used with **pin_mta_worker** thread functions.

Syntax

```
void
pin_mta_get_int_from_pinconf(
          pin_flist_t     *app_info_flistp,
          char            *pinconf_name,
          int32            field,
          int32            flag,
          int32            optional,
          pin_errbuf_t    *ebufp);
```

Parameters

app_info_flistp
A pointer to the PIN_FLD_APPLICATION_INFO substruct in the global flist.

pinconf_name
A pointer to the **pin.conf** file.

field
The name of the field in the **pin.conf** file.

flag
The PIN_FLD_FLAGS value in the PIN_FLD_APPLICATION_INFO substruct is set to this value. For a list of predefined MTA flags, see **pin_mta.h**.

optional
Specifies whether the field is optional or mandatory. If it is mandatory and does not exist in the **pin.conf** file, the error buffer is set.

ebufp
A pointer to the error buffer.

pin_mta_get_str_from_pinconf

This function loads string fields from the **pin.conf** file and sets them to the PIN_FLD_APPLICATION_INFO substruct in the global flist.

> **Note:**
>
> This function is multithread unsafe and cannot be used with **pin_mta_worker** thread functions.

Syntax

```
void
pin_mta_get_str_from_pinconf(
          pin_flist_t     *app_info_flistp,
          char            *pinconf_name,
          int32            field,
          int32            optional,
          pin_errbuf_t    *ebufp);
```

Parameters

app_info_flistp
A pointer to the PIN_FLD_APPLICATION_INFO substruct in the global flist.

pinconf_name
A pointer to the **pin.conf** file.

field
The name of the field in the **pin.conf** file.

optional
Specifies whether the field is optional or mandatory. If it is mandatory and does not exist in the **pin.conf** file, the error buffer is set.

ebufp
A pointer to the error buffer.

pin_mta_global_flist_node_get_no_lock

This function gets the global flist field and does not lock the field.

> **Note:**
>
> This function is multithread unsafe and cannot be used with **pin_mta_worker** thread functions.

Syntax

```
pin_flist_t*
pin_mta_global_flist_node_get_no_lock(
          pin_fld_num_t     field,
          pin_errbuf_t     *ebufp);
```

Parameters

field
The global flist field; for example, PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_global_flist_node_get_with_lock

This function gets the global flist field and sets the lock to prevent access to the field. Used together with **pin_mta_global_flist_node_release**.

Syntax

```
pin_flist_t*
pin_mta_global_flist_node_get_with_lock(
          pin_fld_num_t     field,
          pin_errbuf_t     *ebufp);
```

Parameters

field
The global flist field; for example, PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_global_flist_node_put

This function puts the global flist field with the specified flist.

Syntax

```
void
pin_mta_global_flist_node_put(
          pin_flist_t      *in_flistp,
          pin_fld_num_t     field,
          pin_errbuf_t     *ebufp);
```

Parameters

in_flistp
A pointer to the flist to set in the global flist.

field
The global flist field to set; for example, PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_global_flist_node_release

This function releases the lock from the global flist field. Used together with **pin_mta_global_flist_node_get_with_lock**.

> **Note:**
>
> Do not use this function to release locks unless the locks were set using **pin_mta_global_flist_node_get_with_lock**.

Syntax

```
void
pin_mta_global_flist_node_release(
          pin_fld_num_t     field,
          pin_errbuf_t     *ebufp);
```

Parameters

field
The global flist field; for example, PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_global_flist_node_set

This function sets the global flist field with the specified flist.

Syntax

```
void
pin_mta_global_flist_node_set(
          pin_flist_t     *in_flistp,
          pin_fld_num_t    field,
          pin_errbuf_t    *ebufp);
```

Parameters

in_flistp
A pointer to the flist to set in the global flist.

field
The global flist field to set; for example, PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_init_app

This function, called at application initialization, is a hook to implementing functionality that is required before the main search execution.

Syntax

```
void
pin_mta_init_app(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_init_search

This function prepares the search flist.

This function is called at search initialization. It prepares the search flist that is passed as the input flist to the search opcodes. The search flist is prepared according to the search opcode input flist specification. This function can also be used to update the search flist for subsequent searches.

> **Note:**
>
> The select query should be designed to fetch a new set of records each time it is run, otherwise **pin_mta_search** will return the same records each time, causing your MTA application to stop responding.

One way to accomplish this is to update the objects in the database during processing by the worker threads so that these objects are not returned by the select query when it is run again. For example, suppose a billing application uses the following select query to process accounts that have not been billed:

```
select bill_obj_id0 from account_t where actg_next_t <= current_time
```

The application updates the account by setting **actg_next_t** to the next cycle after the account is processed. When **pin_mta_search** is run again, the select query returns a new set of accounts that have not been billed. Accounts that have already been processed do not meet the search criteria, therefore they are not returned in the result set.

Another way to ensure that new records are fetched each time would be to use **order by** in your select query so that the returned results are ordered. For example:

```
select bill_obj_id0 from account_t where actg_next_t <= current_time order by poid_id0
```

When you run the query again, start from the previous search maximum **poid_id**. For example:

```
select bill_obj_id0 from account_t where actg_next_t <= current_time and poid_id >
previous_maximum_poid_id order by poid_id0
```

This method does not require a database update as does the previous example; however, depending on the number of records being ordered, there might be a performance impact.

Syntax

```
void
pin_mta_init_search(
          pin_flist_t     *app_flistp,
          pin_flist_t    **search_flistpp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information from the global flist substruct PIN_FLD_APPLICATION_INFO.

search_flistpp
A pointer to a pointer to a search flist.

> **Note:**
>
> The search flist allocated in this function is set to the PIN_FLD_SEARCH_FLIST substruct in the global flist.

ebufp
A pointer to the error buffer.

pin_mta_job_done

This function performs functions required at application job completion.

This function is called after all worker threads have finished processing the jobs assigned to them. It is a hook for implementing functionality that is required, such as validation and logging.

Syntax

```
void
pin_mta_job_done(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_main_thread_pcm_context_get

This function gets the main thread context. This context can be reused by the main thread callback functions only.

> **Note:**
>
> This function is multithread unsafe and cannot be used with **pin_mta_worker** thread functions.

Syntax

```
pcm_context_t *
pin_mta_main_thread_pcm_context_get(
          pin_errbuf_t     *ebufp);
```

Parameters

ebufp
A pointer to the error buffer.

pin_mta_post_config

This function performs post-configuration functions.

This function is called after application configuration. It is a hook to implement custom functions that are required after configuration, such as validation, providing results from the configuration policy opcode hook, and logging.

Syntax

```
void
pin_mta_post_config(
          pin_flist_t      *param_flistp,
          pin_flist_t      *app_flistp,
          pin_errbuf_t     *ebufp);
```

Parameters

param_flistp
A pointer to the flist containing information about the command-line parameters.

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_post_exit

This function performs functions required before the application shuts down.

This is the last function called when the application is about to exit. It is a hook for implementing functions that are required, such as validation of results from the application exit policy opcode hook and logging.

Syntax

```
void
pin_mta_post_exit(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_post_init_app

This function performs post-application initialization functions.

This function is a hook for implementing functionality that is required after initialization, such as validation of results from the initialization policy opcode hook and logging.

Syntax

```
void
pin_mta_post_init_app(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_post_init_search

This function performs post-search flist preparation functions.

This function is a hook for implementing functionality that is required after the search flist is prepared, such as validation of results from the search initialization policy opcode hook and logging.

Syntax

```
void
pin_mta_post_init_search(
          pin_flist_t     *app_flistp,
          pin_flist_t     *search_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

search_flistp
A pointer to the flist containing the search flist from global flist substruct PIN_FLD_SEARCH_FLIST.

ebufp
A pointer to the error buffer.

pin_mta_post_job_done

This function performs post-application job-completion functions.

This function is called after all worker threads have finished processing the jobs assigned to them. It is a hook for implementing functionality that is required, such as validation of results from the application job-completion policy opcode hook and logging.

Syntax

```
void
pin_mta_post_job_done(
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_post_tune

This function performs post-search functions for preprocessing search results.

This function is a hook for implementing functions that are required after the search results have been preprocessed, such as validation of the results from search results tuning policy opcode hook and logging.

Syntax

```
void
pin_mta_post_tune(
          pin_flist_t     *app_flistp,
          pin_flist_t     *srch_res_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

srch_res_flistp
A pointer to the flist containing the search results flist received from the global flist substruct PIN_FLD_SEARCH_RESULTS.

ebufp
A pointer to the error buffer.

pin_mta_post_usage

This function displays the help text prepared by **pin_mta_usage** and the usage policy opcode hook.

Syntax

```
void
pin_mta_post_usage(
          pin_flist_t     *param_flistp,
          pin_flist_t     *app_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

param_flistp
A pointer to the flist containing information about the command-line parameters.

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

ebufp
A pointer to the error buffer.

pin_mta_post_worker_exit

This function exits all worker threads.

This function is called when the application is about to exit and all worker threads must exit. This function is a hook for implementing functions that are required, such as logging.

Syntax

```
void
pin_mta_post_worker_exit(
          pcm_context_t     *ctxp,
```

```
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_post_worker_init

This function performs post-worker thread initialization functions.

This function is called for each thread at thread startup. It is a hook for implementing functions that are required after worker thread initialization, such as validation of results from the worker thread initialization policy opcode hook and logging.

Syntax

```
void
pin_mta_post_worker_init(
          pcm_context_t     *ctxp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_post_worker_job

This function performs post-worker thread job preparation functions.

This function is a hook for implementing functions that are required, such as validation of results from the worker thread policy opcode hook and logging.

Syntax

```
void
pin_mta_post_worker_job(
          pcm_context_t     *ctxp,
          pin_flist_t       *srch_res_flistp,
          pin_flist_t       *op_in_flistp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

srch_res_flistp
A pointer to the flist containing a subset of the global search results assigned to a worker thread.

op_in_flistp
A pointer to the flist containing the main opcode input flist.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_post_worker_job_done

This function performs post-worker job completion functions.

This function is a hook for implementing functions that are required, such as validation or processing of results from the worker thread policy opcode hook.

Syntax

```
void
pin_mta_post_worker_job_done(
          pcm_context_t     *ctxp,
          pin_flist_t       *srch_res_flistp,
          pin_flist_t       *op_in_flistp,
          pin_flist_t       *op_out_flistp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

srch_res_flistp
A pointer to the flist containing a subset of the global search results assigned to a worker thread.

op_in_flistp
A pointer to the flist containing the main opcode input flist.

op_out_flistp
A pointer to the flist containing the main opcode output flist.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_tune

This function preprocesses search results.

This function is called after the main search execution for preprocessing the search results. The search results can be modified before the results are distributed to the worker threads.

Syntax

```
void
pin_mta_tune(
          pin_flist_t     *app_flistp,
          pin_flist_t     *srch_res_flistp,
          pin_errbuf_t    *ebufp);
```

Parameters

app_flistp
A pointer to the flist containing application information received from the global flist substruct PIN_FLD_APPLICATION_INFO.

srch_res_flistp
A pointer to the flist containing the search results flist received from the global flist substruct PIN_FLD_SEARCH_RESULTS.

ebufp
A pointer to the error buffer.

pin_mta_usage

This function creates a help text message.

This function is called when the user requests help using the **-help** parameter or when an error occurs during application configuration. This function is a hook to prepare help text messages.

> **Note:**
>
> You must set the PIN_FLD_DESCR field in PIN_FLD_EXTENDED_INFO substruct in the global flist to the help text so that it can be accessed by the usage policy opcode for customization and by **pin_mta_post_usage** for displaying the message.

Syntax

```
void
pin_mta_usage(
          char      * prog);
```

Parameters

prog
The application name.

pin_mta_worker_exit

This function exits all worker threads.

This function is called when the application is about to exit and all worker threads must exit. This function is a hook for implementing functions that are required, such as logging.

Syntax

```
void
pin_mta_worker_exit(
          pcm_context_t      *ctxp,
          pin_flist_t        *ti_flistp,
          pin_errbuf_t       *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_worker_init

This function performs thread initialization.

This function is called for each worker thread at thread startup. It is a hook for implementing functions that are required at worker thread initialization.

Syntax

```
void
pin_mta_worker_init(
          pcm_context_t      *ctxp,
          pin_flist_t        *ti_flistp,
          pin_errbuf_t       *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_worker_job

This function performs functions required at worker thread job preparation.

This function is called every time a worker thread receives work for any preprocessing of the search results before the results are passed to the main opcode to be processed.

> **Tip:**
>
> This function provides the same functionality as **pin_mta_tune**. Because of parallel processing, preprocessing search results in the worker threads is more efficient when BRM is installed on a multiple-CPU host machine.

Syntax

```
void
pin_mta_worker_job(
          pcm_context_t     *ctxp,
          pin_flist_t       *srch_res_flistp,
          pin_flist_t      **op_in_flistpp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

srch_res_flistp
A pointer to the flist containing a subset of the global search results assigned to a worker thread.

op_in_flistpp
A pointer to a pointer to the main opcode input flist.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_worker_job_done

This function performs functions that are required after the main opcode has processed the batch of search results.

This function is a hook to implementing functions that are required, such as the validation of main opcode results.

Syntax

```
void
pin_mta_worker_job_done(
          pcm_context_t     *ctxp,
          pin_flist_t       *srch_res_flistp,
          pin_flist_t       *op_in_flistp,
          pin_flist_t       *op_out_flistp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

srch_res_flistp
A pointer to the flist containing a subset of the global search results assigned to a worker thread.

op_in_flistp
A pointer to the flist containing the main opcode input flist.

op_out_flistp
A pointer to the flist containing the main opcode output flist.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

pin_mta_worker_opcode

This function runs the main opcode.

This function is called by the worker threads to run the main opcode to process the search results. Worker threads call this function for every batch of work they receive.

Syntax

```
void
pin_mta_worker_opcode(
          pcm_context_t     *ctxp,
          pin_flist_t       *srch_res_flistp,
          pin_flist_t       *op_in_flistp,
          pin_flist_t      **op_out_flistpp,
          pin_flist_t       *ti_flistp,
          pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to the PCM context.

srch_res_flistp
A pointer to the flist containing a subset of the global search results assigned to a worker thread.

op_in_flistp
A pointer to the flist containing the main opcode input flist.

op_out_flistpp
A pointer to a pointer to the flist containing the main opcode output flist.

ti_flistp
A pointer to the flist containing thread information.

ebufp
A pointer to the error buffer.

42

Creating Client Applications by Using Java PCM

Learn how to create Java client applications that communicate with Oracle Communications Billing and Revenue Management (BRM) by using the Java Portal Communication Module (Java PCM) Application Programming Interface (API).

Topics in this document:

- About Using the Java PCM API
- Using the Java PCM API
- About Creating Client Applications by Using the Java PCM API
- Specifying a Timeout Value for Requests
- Using the Asynchronous PCP Mode in Java PCM Client Libraries
- Setting Global Options
- Running the jnap Utility
- About the Sample Program

See also:

- About Customizing BRM
- Understanding the PCM API

About Using the Java PCM API

You use the classes and their methods in the Java PCM API to write Java client APIs, see *Java PCM API*.

To use the Java PCM package, you must have the following skills and experience:

- Experience in developing Java applications
- A good understanding of the BRM architecture and the following concepts:
  - PCM opcodes
  - Portal Information Network (PIN) libraries
  - flists (field lists)
  - Context (**PortalContext**)
  - Buffers (**Buffer**, **FileBuffer**, **ByteBuffer**)
  - Fields (**Fields**)
  - Portal object IDs (POIDs)

For information on BRM architecture and concepts, see the following topics:

- BRM System Architecture

- Understanding the PCM API
- Understanding the BRM Data Types
- Understanding Flists

Software Requirements

Use the Java PCM API on a supported version of the Java Development Kit (JDK). See *BRM Compatibility Matrix*.

About the Java PCM API and the C API

The Java PCM API consists of a set of Java classes that represent the BRM C data structures, such as flists, fields, context, and POIDs, that are defined in the **pcm.h** file.

For information on BRM data types, see "Understanding the BRM Data Types".

For information on flists, see "Understanding Flists".

The Java API differs from the C API in the following ways:

- Timestamps and strings are represented by the **Date** and **String** Java classes. There is a separate array class for flist arrays (**SparseArray**).
- The information stored in the C error buffer is part of **EbufException** in the Java PCM package.
- In the Java PCM package, opcodes are constants in the **PortalOp** class and are named without the **PCM_OP** prefix. For example, **PCM_OP_READ_FIELD** in the C API is **PortalOp.READ_FIELD** in the Java PCM package.
- For type safety, field names are provided as classes.

  Field names follow the Java class-naming conventions and use mixed case without the underscores. The Java field classes in the Java PCM package use the C **#define** name without the **PIN**_ prefix. For example, PIN_FLD_NAME_INFO in C becomes **FldNameInfo** in Java.
- Field instances are shared to improve performance; therefore, you pass a field to a method by using the following syntax:

  ```
  obj.method(FldNameInfo.getInst());
  ```

  where *obj* is the object that calls the *method* method.

Using the Java PCM API

Follow these steps when using the Java PCM API:

- Make sure that the **pcm.jar** and **pcmext.jar** files are in your **CLASSPATH**.
- Include the following import statements in the Java files that use PCM classes:
  - **import com.portal.pcm.*;**
  - **import com.portal.pcm.fields.*;**
- When you run a Java program that communicates with BRM, make sure the **Infranet.properties** files and the custom **.CLASS** files are in the **CLASSPATH**.

About Creating Client Applications by Using the Java PCM API

When you create client applications that use the Java PCM API, you can use either the synchronous mode or the asynchronous mode of handling requests.

About Synchronous and Asynchronous Modes

In BRM, when you create a client application to process requests in synchronous mode, the application sends a request and waits for a response from the Connection Manager (CM) before it sends the next request. If there is an error on the server side in processing the request, the client application may have to wait indefinitely. To address this, you can set a timeout value for the request your client application sends to the CM. If the CM does not respond within the time specified in the request, the PCP connection layer returns an error message to the client application and closes the connection. See "Specifying a Timeout Value for Requests".

Alternately, when you create a client application to process requests in asynchronous mode, it sends a request to BRM and then, instead of waiting for a response to the request, the client application continues to perform other operations. Throughput is improved in this mode because client applications can make multiple requests. When the response for a request becomes available, the client application retrieves the response and proceeds further with its processing of that request. See "Using the Asynchronous PCP Mode in Java PCM Client Libraries".

Actions Performed by BRM Java Client Applications

A BRM Java client application does the following:

1. Opens a connection.
2. Clears the error buffer.
3. Performs PCM operations.
4. Checks for errors.
5. Closes the connection.

For more information, see "Creating a Client Application in C".

When you call an opcode, you need to create an flist and pass it as an input. See the input flist specification in the opcode descriptions for information on the structure of the flist.

For information on how to create an flist, see "Flist Creation Samples".

Opening a PCM connection

You open a connection to BRM by using the Java PCM API, all communication is performed through the **PortalContext** class. You can use one of the following methods to open a connection:

- Call the **open()** method on the context class with all the login information needed, including the host name and the port number.
- Use the **connect()** method, if you want your program to log in automatically to BRM. You must store all the necessary login information in the **Infranet.properties** file and make sure that file is in the **CLASSPATH**.

For information on creating an **Infranet.properties** file, see "Setting Global Options".

> **Note:**
>
> Use the **Infranet.properties** file for configuring Java applications instead of the **pin.conf** file used by C applications.

If your custom application supports multiple database schemas, you must use the database number returned by the PCM_CONNECT() or PCM_CONTEXT_OPEN() opcodes for all transactions within the context you open. The **open()** and **connect()** methods call these opcodes to open a connection to BRM.

To open a transaction on a specific database schema, use the **opcode** method within the **PortalContext** class to run the opcode PCM_OP_TRANS_OPCODE with the schema POID specified in the input flist. For more information, see PCM_OP_TRANS_OPEN.

Using Custom Fields in Java Applications

You use Storable Class Editor to create custom fields and to generate Java source files that you compile into classes. See "Creating, Editing, and Deleting Fields and Storable Classes".

You can use custom fields in flists in your Java code in the same manner as BRM fields. Before class names are created from the fields, the prefix **PIN_** is removed from the BRM fields and custom field names are used as they are for class names.

To create and use custom fields in the Java PCM package:

1. Start Storable Class Editor and create your storable classes and fields.
2. From the **File** menu, choose **Generate Custom Fields Source** to create source files for your custom fields.

   Storable Class Editor creates a C header file called **cust_flds.h**, a Java properties file called **InfranetPropertiesAdditions.properties,** and a Java source file for each custom field.
3. For each Java application that will use these fields, copy the contents of the **InfranetPropertiesAdditions.properties** file and paste it into each application's **Infranet.properties** file.
4. In the directory where Storable Class Editor created the Java source files, compile the source files:

   ```
   javac -d . *.java
   ```
5. Package the class files created in step 4 into a **JAR** file:

   ```
   jar cvf filename.jar *.class
   ```
6. In the CLASSPATH, add the location of the **JAR** file.

Creating Custom Storable Classes

You use Storable Class Editor to create custom storable classes. See "Creating, Editing, and Deleting Fields and Storable Classes" for instructions. After you commit the storable classes in Storable Class Editor, create the object in Java:

```
Poid aPoid = new Poid(database, -1, "/customClass");
inFlist.set(FldPoid.getInst(), aPoid);
```

```
/*... set other fields*/
FList outFList = ctx.opcode(PortalOp.CREATE_OBJ, inFlist);
```

Calling Custom Opcodes

To call custom opcodes:

1. Write and define your custom opcodes on the server:

   ```
   #define custom_opcode1 10001
   ```

2. Call the appropriate method of the **PortalContext** class in Java. The method you call depends on the opcode processing mode:
   - Synchronous mode: Use the **opcode** method. See "Using Synchronous Mode for Opcode Processing".
   - Asynchronous mode: Use the **opcodeSend** method. See "How Asynchronous Mode for Opcode Processing Works".

For more information on creating custom opcodes, see "Defining New Opcodes".

For maintainability, you can create a class with all your opcodes, as shown in the following example:

```
class CustomOpcode{
      public static final int OPCODE1 = 10001;
      public static final int OPCODE2 = 10002;
      /*more custom opcodes here*/
}
```

Using Synchronous Mode for Opcode Processing

To create client applications that use the synchronous mode of opcode processing, call the **opcode** method of the **PortalContext** class. For example, the following syntax would be used to call the custom opcode, *OPCODE1*:

```
ctx.opcode(CustomOpcode.OPCODE1, inFlist);
```

Internally, the **opcode** method sends the information on the custom opcode (*OPCODE1*) to the BRM server and waits for a response. Your client application is blocked during this time interval and can resume further action only after the output is received from the execution of *OPCODE1*.

Getting a Text Format of an Flist

When debugging a client application, it is often useful to read a text representation of an flist. The **flist** API provides the following methods:

- **toString()** method for general purposes
- **dump()** method to display on standard output

Handling Exceptions

Any call that causes a PCM-related error throws an **EbufException** exception. If partial information was available in an flist from an opcode call, you can retrieve it from the **EbufException** exception in the Java package.

> **Tip:**
>
> For efficiency, catch exceptions at the highest possible level.

For help on the available options and usage, type **?** at the prompt.

> **Note:**
>
> All the information available in the C error buffer is available in the Java **EbufException** exception when that exception is caught.

Logging Errors and Messages

The **ErrorLog** class contains the API for logging status, errors, and other messages to a file or buffer. You can access the default log through the **PortalContext** class. Always use the log to ensure that you get the expected output.

Even when the error log function is used from an applet in a Web browser, where the **Security Manager** denies access to the file system, a log is generated in a buffer. You must access and display the log from the applet.

Infranet.properties has several options for controlling debugging, including automatic logging of all **EbufException** exceptions. For more information on the options, see "Setting Global Options".

The Java log function does not have an indefinite list of arguments. You must use string concatenation to form a simple string message. For example:

```
errorlog.log(ErrorLog.Error, "Here's the error and flist:\n" + error.toString() + "\n" +
flist.toString());
```

Specifying a Timeout Value for Requests

You can specify a timeout value for each request to the CM in your client application. For more information on implementing timeout values, see "Implementing Timeout for Requests in Your Application".

To specify a timeout value in milliseconds for a connection, pass the PIN_FLD_TIMEOUT_IN_MS field in the input flist to the **PortalContext** class constructor or the **open()** or **connect()** methods of the **PortalContext** class. The PIN_FLD_TIMEOUT_IN_MS value is applicable after the client connects to the server. Before the connection occurs, this setting is not effective. It does not report a timeout if the client cannot connect to the server at all.

For more information on the **PortalContext** class, see *BRM PCM Java API Reference*.

The timeout value you specify applies to all the opcodes called during that open session and overrides the value in the client properties file. You must ensure that your client application handles the timeout, closes its connection to the CM by calling PCM_CONTEXT_CLOSE, and cleans up the transaction context.

> **Note:**
>
> When the timeout occurs, the CM does not provide any feedback about the success or failure of the request it received. When the CM detects the closed connection, it rolls back the ongoing transaction and shuts down.

Using the Asynchronous PCP Mode in Java PCM Client Libraries

When a Java client application uses the Java PCM library and calls a BRM opcode, the PCP connectors forward the request and wait for the responses from BRM. Asynchronous mode of processing enables client applications to handle multiple requests and improve the throughput of opcode calls.

When you create a client application that uses asynchronous mode processing, you can call the **opcodeSend** method for as many opcode processing requests as required. As the responses from the execution of these opcode operations become available, your client application can call the **opcodeReceive** method to receive the responses and act upon them.

Asynchronous mode processing can be used for base opcodes and custom opcodes.

The following classes in the Java PCM client libraries enable you to create client applications that make use of the asynchronous mode:

- **PCPSelector**, the PCP connection's socket channel selector. See "About PCPSelector".
- **PortalContextListener**, the abstract listener. See "About PortalContextListener".

About PCPSelector

PCPSelector is the PCP connection's socket channel selector that uses Java NIO mechanism during the asynchronous mode. (NIO stands for New I/O, a set of nonblocking APIs for Java programming language that offer features for intensive Input/Output operations.)

This Java class provides a Java method named **process** and also internally calls other interfaces responsible for:

- Registering the socket channels with the selector.
- Changing the socket channel's interest to READ (data available) operation after the opcode is sent.
- Sending the notification to **PortalContextListener** once the data has arrived.

The client application that you create must extend the **PCPSelector** class and implement its **Runnable** interface. To take full advantage of the multithreaded selector capabilities of the **PCPSelector** class, you must call the **process** method in your implementation's **run** method, as shown in the following example:

```
public class SDPPCPSelector extends PCPSelector implements Runnable  {

       private static SDPPCPSelector instance = null;
       private SDPPCPSelector() throws IOException {
               super();
              }

       public static synchronized SDPPCPSelector getInstance() throws IOException
       {
        if (instance == null) {
```

```
                                instance = new SDPPCPSelector();
                                }
         return instance;
        }

         public void run(){
                          System.out.println("inside Run of SDP PCPSelector");
                          process();
                         }
}
```

Below, an instance of the SDPPCPSelector class is created. It can be used to spawn multiple threads.

```
PCPSelector pcpSelector = SDPPCPSelector.getInstance();
Thread t = new Thread((Runnable) pcpSelector);
t.start();
```

For more information on **PCPSelector**, see *PCM Java API Reference*.

About PortalContextListener

PortalContextListener is an abstract listener. It forces the client application to implement the notification-handling interface that is used when opcode operations are performed in an asynchronous mode. The **handlePortalContextEvent** method of **PortalContextListener** supplies the **PortalContext** object with the related event that has occurred recently.

The PCM library supports notifications for the following:

- EVENT_READ_DATA_AVAILABLE
- EVENT_CHANNEL_CLOSED

Client applications can extend **PortalContextListener** and use the child class to register their interest in these two events by calling the **addListener** method of **PCPSelector**.

For example:

```
public class ClientListener extends PortalContextListener {
ClientListener () {
       pcpSelector = SDPPCPSelector.getInstance();
       pcpSelector.addListener(this,EventType.EVENT_READ_DATA_AVAILABLE);
       pcpSelector.addListener(this,EventType.EVENT_CHANNEL_CLOSED);
      }
       public void handlePortalContextEvent(PortalContext portalContext,EventType
eventType) {
// use another thread mechanism to call actual opcodeReceive()
      ...
     }

}
```

For more information on **PortalContextListener**, see *PCM Java API Reference*.

How Asynchronous Mode for Opcode Processing Works

The asynchronous mode for opcode processing works in the following manner:

1. Your client application calls the **opcodeSend** method of the **PortalContext** class. For example code:

   ```
   ctx.opcodeSend(CustomOpcode.OPCODE1, inFlist);
   ```

Internally, the **opcodeSend** method of the **PortalContext** class sends the information on the opcode (*OPCODE1*) to the BRM server, but it does not wait for the response from the opcode operation. Immediately after calling **opcodeSend**, the client application can perform other tasks.

The BRM framework takes care of monitoring the sockets to check if the **PortalContext** data (that is, the response from the opcode operation) is available. When this data is available, the **handlePortalContextEvent** method of the **PortalContextListener** object in your client application is called automatically.

2. When your client application receives the notification through the **handlePortalContextEvent** method, it invokes the **opcodeReceive** method to receive the response of the opcode operation from BRM.

> **Note:**
>
> For better performance, Oracle recommends that you use a separate thread to call the **opcodeReceive** method rather than calling the **opcodeReceive** method directly from the **handlePortalContextEvent** method.

For information on **PortalContext**, see *BRM PCM Java API Reference*.

Creating Client Applications for Asynchronous Mode of Opcode Processing

For the client application to use the asynchronous opcode processing feature in BRM:

- Set up your client applications so that they have multiple threads of the **PCPSelector** object. To ensure asynchronous opcode processing, pass the **PCPSelector** object when you create the **PortalContext** object. For example:

  ```
  PortalContext portalContext = new PortalContext(pcpSelector);
  ```

- Register the client application's interest for specific events by calling the **addListener** method of the **PCPSelector** object.
- Set up the client application to have multiple **PortalContextListener** objects, if necessary. Each **PortalContextListener** object provides a notification mechanism through its **handlePortalContextEvent** method.
- If the client application uses multiple **PortalContextListener** objects, make the readable **PortalContext** object available to any one or all of the **PortalContextListener** objects.

  To make the readable **PortalContext** object available to a *specific* **PortalContextListener**, use the **registerContextToListener** method, as shown in this example:

  ```
  PortalContextListener pListener = new CustomPortalContextListener();
  PortalContextListener qListener = new CustomPortalContextListener();
  PortalContext portalContext = new PortalContext(pcpSelector);
  pListener.registerContextToListener(portalContext);
  ```

  When this sample code runs, one specific **PortalContextListener** object (pListener) is registered with the **PortalContext** object. When data becomes available on the **PortalContext** object and that object is ready to call the **opcodeReceive** method, a notification is sent to that specific **PortalContextListener** object (pListener).

  To make the readable **PortalContext** object available to all **PortalContextListener** objects, omit the **registerContextToListener** method, as in this example:

```
PortalContextListener pListener = new CustomPortalContextListener();
PortalContextListener qListener = new CustomPortalContextListener();
PortalContext portalContext = new PortalContext(pcpSelector);
```

None of the **PortContextListener** objects is registered with the **PortalContext** object (because the **registerContextToListener** method was not called for any **PortContextListener** object). As a result, when data becomes available on the **PortalContext** object and that object is ready to call the **opcodeReceive** method, the notification is sent to all the **PortalContextListener** objects.

- You can set up your client application for multiple opcode processing. For example, the following example issues a second **opcodeSend** request without waiting for the actual response from the first opcode operation:

```
PortalContext portalContext1 = new PortalContext(pcpSelector);
portalContext1.open(login_params_flist);
portalContext1.opcodeSend(opcode_name, input_flist);

PortalContext portalContext2 = new PortalContext(pcpSelector);
portalContext2.open(login_params_flist);
portalContext2.opcodeSend(opcode_name, input_flist);
```

- **PortalContext** provides different APIs that your client application can use to check the status of the socket channel.

For more information on the **PortalContext** class, see *PCM Java API Reference*.

Setting Global Options

Infranet.properties is an optional configuration file that contains entries to control *all* connections from Java applications to BRM. Java PCM looks in the **Infranet.properties** file for information not provided in the login flist. You must include the **Infranet.properties** file in the **CLASSPATH**.

The content of the **Infranet.properties** file conforms to the Java properties file conventions. Options are key-value pairs separated by the equal sign (**=**). For example, **host=ip://test2:11960** and **log.file=mylog.log**.

> **Note:**
>
> You *must* include an entry of the form **type = 1** in the **Infranet.properties** file if that entry is not in the login flist.

Default Entries in the Infranet.properties File

Table 42-1 lists the predefined entries and their types:

Table 42-1 Entries in Infranet.Properties

Entry	Value	Notes
Infranet.login.type	**0** or **1**	Specifies the type of login. A type 1 login requires the application to provide a user name and password. A type 0 login is a trusted login that comes through a CM Proxy, for example, and does not require a user name and password in the properties file.
Infranet.connection	**For a type 1 login:** pcp://*username*:*password*@*hostname*:*port*/*service*:1 **For a type 0 login:** pcp:/*hostname*:*port*/*database_no*/*service*:1 Where: • *username* is the login name to use for connecting to BRM. • **Note**: *username* cannot contain the characters **:** and @. The **/** character is allowed. • *password* is the password for the specified user name. • **Note**: *password* cannot contain the characters **:** and @. The **/** character is allowed. • *hostname* is the name or IP address of the computer running the CM or CMMP. • *port* is the TCP port number of the CM or CMMP on the host computer. The port number must match the corresponding **cm_ports** entry in the CM or CMMP configuration file. • *service* is the service type. The trailing **1** is the POID of the service. • *database_no* is the number assigned to your BRM database when the Oracle Data Manager was installed.	Specifies the full URL to the BRM service. For a type 1 login, the URL must include a user name and password. You must specify the service name and service POID ("1"), but the CM determines the database number. A type 0 login requires a full POID, including the database number.
Infranet.failover.nn	pcp://hostname:port	Specifies the alternative CM hosts that the application can use to connect to BRM if the main host (specified in the connection entry) is unavailable. The user name, password, and service for these alternative hosts are the same as for the main host and is not specified in the failover entries. Failover entries are numbered sequentially, starting with 1.

Optional Entries in the Infranet.properties File

The **Infranet.properties** file can contain other entries. Some of these are based on keys in the Java PCM API, and others are written specifically for an application or tool.

Table 42-2 shows entries from the Java PCM API.

Table 42-2 Optional Entries in Infrant.properties

Entry	Description
Infranet.log.file	The file path. The default is **javapcm.log**.
Infranet.log.logallebuf	Boolean. If **true**, forces all **EbufException** exceptions to be logged automatically.
Infranet.log.level	Specifies how much information the application should log: • **0**: No logging • **1**: Log ERROR messages • **2**: Log ERROR and WARNING messages • **3**: Log ERROR, WARNING, and DEBUG messages
Infranet.pcp.debug.enabled	Boolean. If **true**, enables debug mode.
Infranet.pcp.debug.flags	Specifies what to log: • **0**: Log nothing • **1**: Log errors only • **0x1fff**: Log all messages
Infranet.log.opcodes.enabled	Boolean. If **true**, enables a log that records the input and output flist for every opcode called by all client applications that support this feature.
Infranet.log.opcodes.file	The file path.

Table 42-3 shows entries used for NamedLogs. The *log_name* variable specifies the NamedLog name, such as the application doing the logging.

Table 42-3 Named Log Entries

Entry	Description
Infranet.log.*log_name*.**file**	Full path to the log file for the application. The log file shows errors, warnings, and debugging messages associated with the application, as specified by the **Infranet.log.***log_name*.**level** entry.
Infranet.log.*log_name*.**style**	Specifies the logging control style. The value for this entry can be: • **priority**: Logs messages according to their priority level, specified by the **Infranet.log.***log_name*.**level** entry. • **flag**. Logs messages according to their type, specified by the **Infranet.log.***log_name*.**level** entry.
Infranet.log.*log_name*.**level**	Specifies how much information the application should log. Possible values depend on the log control style specified by the **Infranet.log.***log_name*.**style** entry. For **priority** style, set the level as a decimal value. All messages with a priority level lower than this level are logged. (Low number = high priority) For **flag** style, set the level to one of these values: • **0**: no logging • **1**: log ERROR messages • **2**: log ERROR and WARNING messages and • **3**: log ERROR, WARNING, and DEBUG messages
Infranet.log.*log_name*.**logallebuf**	Boolean. If **true**, forces all **EbufException** exceptions to be logged automatically.
Infranet.log.*log_name*.**name**	Specifies the name of the log where all messages of a specific type are written to.

Table 42-3 (Cont.) Named Log Entries

Entry	Description
Infranet.log.*log_name***.enabled**	Enables or disables NamedLog logging. To disable logging, set to **f**, **n**, or **0**.

Table 42-4 shows an entry that you can use to display a list of hosts:

Table 42-4 Entries used to List Hosts

Entry	Description
Infranet.*application_name***.host**.*nn*	A list of hosts, (for example, **host.1**, **host.2**), and so on, displayed when a user opens the application. The user can select to connect to any host in the list. Host entries, each consisting of a pair of values for *hostname* and *port*, are numbered sequentially, starting with *1*. Other connection information, such as service, comes from the standard **Infranet.connection** entry.

Example Infranet.properties File

```
infranet.connection=pcp://sommachine:11960/service/admin_client 1
infranet.login.type=1
infranet.log.file=fullOutput.log
infranet.log.logallebuf=true
infranet.log.level=3
infranet.pcp.debug.flags=0x3FFFFFFF
infranet.pcp.debug.enabled=true
```

Controlling Opcode Logging from a Client Application

An opcode log contains the input and output flist for every opcode called by the following applications:

- Payment Center
- Permissioning Center
- Revenue Assurance Center
- Suspense Management Center
- Zone Mapper

You can dynamically turn opcode logging on and off for an individual application from the application itself, independent of the global opcode logging settings in the **Infranet.properties** file and without having to restart the application.

> **Note:**
>
> If you do not define an **Infranet.log.opcodes.file** entry in the **Infranet.properties** file and turn logging on using this procedure, the default is **opcodes.log**.

To turn opcode logging on or off:

1. In the application, click **Help** then **About**, and then click **System Information**.
2. In the System Information dialog box, press CTRL+SHIFT and click three times on the label above the table to open the Opcode Logging dialog box.
3. Select or clear the **Log Opcodes** check box for an application to turn logging on or off.

Running the jnap Utility

The Java package includes the **jnap** test utility. You use **jnap** to test the database connection, load flists from files, use the flists as input when calling opcodes on the server, and display output flists.

The **jnap** utility is similar to and provides a subset of the functionality of **testnap**, the utility used for testing applications written using the BRM C API. You can use many of the same commands in **jnap** that you use in **testnap**. Unlike **testnap**, however, **jnap** uses Java PCM to communicate with BRM.

When you start **jnap**, you must include the **Infranet.properties**, **pcm.jar**, and **pcmext.jar** files in the CLASSPATH. In Windows, for example, if the **Infranet.properties** file is in the current directory and the **pcm.jar** and **pcmext.jar** files are in **C:\Portal\jars**, you would enter:

```
java -classpath .;c:\Portal\jars\pcm.jar;c:\Portal\jars\pcmext.jar com.portal.pcm.jnap
```

If you plan to use **jnap** frequently, you can also set CLASSPATH as an environment variable.

Getting Help with jnap

You can get command-line help for **jnap** by entering **help** at the prompt. You see a list of valid commands and variables:

```
jnap> help
jnap command set::
r     <file> <bufnum>      - read flist from file into buffer
i     <bufnum>             - insert flist from STDIN into  buf
d     <flist>              - displays flist
w     <buf> <file>         - save buf to file
l                          - list buf nums used
login                      - login using values in Infranet.properties
login <flist>              - login using specified flist
logout                     - logout and close context
xop   <op> <flags> <flist> - execute op and set 'incoming' buf
q                          - quit
loop  <flist>              - stream out/in flist
pcpdebug  <dbgflags>       - set pcp debug flags
h or ? or help             - displays this help

where
<flist> :: <file> | <buf>
<op>    :: <num>  | <opname>
<opname>:: READ_OBJ | commit_customer | TRANS_OPEN ...
<buf>   :: <bufnum>  | incoming
<flags> :: <number> | <flags>'+'<flag>
<flag>  :: calc|meta|rev|count|add|poid|rdres|nores|ro|rw|lock
<dbgflags> :: <number> | <dbgflags>'+'<dbgflag>
<dbgflag>  :: none|errors|flist_req|flist_resp|read|write|
              read_wh|write_wh|read_dump|
             write_dump|open|connect|trans|op|all
```

Example of Using jnap

This section includes an example of using **jnap**. In this example, you create an flist and then use it as input to the PCM_OP_ACT_TEST_LOOPBACK opcode. This opcode tests the database connection and returns the same flist as output.

1. Use a text editor to create a simple flist and save it as **flist1**.

```
0 PIN_FLD_PROGRAM_NAME      STR [0] "Example"
0 PIN_FLD_POID              POID [0] 0.0.0.1  -1 0
0 PIN_FLD_NAME               STR [0] "Test"
```

2. Read **flist1** into buffer 1.

```
jnap> r flist1 1
```

3. To ensure that the flist was saved, display the contents of buffer 1.

```
jnap> d 1
```

4. Log in to the database.

```
jnap> login
```

5. Run the PCM_OP_ACT_TEST_LOOPBACK opcode with the **xop** command. Include the opcode without the PCM_OP_ACT prefix, **0** for the opcode flag, and **1** for the buffer you will use for the input flist.

```
xop TEST_LOOPBACK 0 1
```

   The output of the opcode is displayed. In this case, the output is the same as the input.

6. Log out.

```
jnap> logout
```

7. Quit **jnap**.

```
jnap> q
```

About the Sample Program

For a sample program for creating a client application, see **SampleApp.java.**

This program creates a customer account by performing these steps:

- Opening a database channel
- Retrieving a package list
- Adding customer information to the account
- Creating the customer account
- Closing the database channel

To run **SampleApp.java**, first edit the **Infranet.properties** file to change the entry *your server* to the host name of the computer where BRM is installed. Then include the file in your CLASSPATH. For an example of the **Infranet.properties** file, see "Example Infranet.properties File".

43

Creating Client Applications by Using Perl PCM

Learn about the Perl extension to the Oracle Communications Billing and Revenue Management (BRM) Portal Communications Module (PCM) library.

Topics in this document:

- About the Perl API
- Differences between the Perl API and the C API
- Guidelines for Using the Pcmif Module
- Performing PCM Operations

See also:

- About Customizing BRM
- Understanding the PCM API

About the Perl API

The Perl extension to the PCM library, pcmif, allows you to use Perl scripts to perform the following PCM operations:

- Connect to PCM.
- Perform PCM opcode operations, such as creating an object, searching for objects, deleting an object.
- Convert flists (field lists) between text and binary formats.
- Generate error reports.

For more information, see the following documents:

- For a description of the Application Programming Interface (API), see "Perl Extensions to the PCM Libraries" in *BRM Developer's Reference*.
- For sample scripts, see "Example Perl Scripts" in *BRM Developer's Reference*.
- For the latest information on the Perl extension, see **pod2text** (text format) or **pod2html** (HTML format) in the *BRM_SDK_home***/lib/pcmif.pm**.

Differences between the Perl API and the C API

The API functions in pcmif are wrappers for a subset of the underlying C functions. You can use the Perl API functions to perform any BRM PCM opcode operation.

The pcmif API functions use the following naming conventions:

- If the C function and its corresponding Perl function use exactly the same arguments, they have the same name.

- If the arguments are different, the Perl equivalent to the C function **pin_function()** is named **pin_perl_function()** to differentiate them.

Guidelines for Using the Pcmif Module

To perform PCM operations by using Perl scripts, follow these guidelines:

- Make sure your scripts include the following information:

  The path to the Perl binaries in the first line:

  ```
  #!/BRM_SDK_home/perl/bin/perl;
  ```

  The Perl module to use:

  ```
  use pcmif;
  ```

- Use Perl 5.0004, included with BRM.

  If you use a different version of Perl, make sure you specify the path to the pcmif files in the Perl command line or by including the **use lib <path>** line in the beginning of your Perl scripts.

- Add the class prefix **pcmif::** to the functions, which specifies that these functions are from the pcmif package. For example:

  ```
  pcmif::pcm_perl_new_ebuf();
  ```

- Make sure you use the *PerlScriptName*.**bat** instead of the *PerlScriptName***.pl**.

Performing PCM Operations

To write Perl scripts that perform BRM operations, follow these guidelines:

- Connect to PCM by opening a PCM connection and a PCM context to the BRM.

  See "Connection Functions" in *BRM Developer's Reference* for a description of the API.

  If your custom application supports multiple database schemas, you must use the database number returned by PCM_CONNECT() or PCM_CONTEXT_OPEN() for all transactions within the context you open.

- Check for errors after each action.

  See "Error-Handling Functions" in *BRM Developer's Reference* for a description of the API.

- Convert flists between text and binary formats:
  - Convert strings that you create in your Perl script to flist format before performing PCM operations.

    For example, if you use a "here" document to assign an flist string to a variable in your Perl scripts, convert the variable to flist format before searching for an object by using PCM_OP_SEARCH.

  - Convert flists to string format before using them in Perl functions. For example, to pass an flist as input to Perl functions such as string matching, first convert the flist to a string.

    See "Flist Conversion Functions" in *BRM Developer's Reference* for a description of the flist conversion functions.

  - Perform the PCM operations.

    See Error-"Handling Functions" in *BRM Developer's Reference* for a description of the API.

- Delete the flists and error buffers you no longer need.

  See "Error-Handling Functions" and "Flist Conversion Functions" in *BRM Developer's Reference* for a description of the functions to use.

- Disconnect from PCM by closing the PCM context.

  See "Connection Functions" in *BRM Developer's Reference* for a description of the API.

44

Creating Client Applications by Using PCM C++

Learn how to create C++ client applications that communicate with an Oracle Communications Billing and Revenue Management (BRM) system by using the Portal Communication Module (PCM) C++ Application Programming Interface (API).

Topics in this document:

- About PCM C++
- Understanding PCM C++ Concepts
- Using the PCM C++ API
- Debugging PCM C++ Programs
- Troubleshooting

> **Note:**
>
> Before using PCM C++, read "Comparison of the PCM C++ and PCM C APIs".

About PCM C++

Use the classes and their member functions in the PCM C++ API to write C++ client applications that communicate with BRM. PCM C++ is a set of wrappers around the PCM C client library. The C++ classes represent the BRM C data structures. These structures, including flists, fields, context, and POIDs, are defined in the **pcm.h** file.

However, C++ provides several advantages over C: improved runtime memory, type checking, compile time, and reliability. C++ provides hooks for assertions in debugging. If programmed properly, smart pointers can make memory management easier for programmers. In addition, C++ allows object oriented programming, a more powerful approach than is possible in C. PCM C++ allows you to take advantage of these C++ capabilities.

For information on BRM data types, see "Understanding the BRM Data Types".

For information on flists, see "Understanding Flists".

Skills Required

To use the PCM C++ package, you must have the following skills and experience:

- Experience in developing C++ applications and an understanding of smart pointers.

  For information on PCM C++, see "Using the PCM C++ API".
- A good understanding of BRM architecture and the following concepts:
  - PCM opcodes

- Portal Information Network (PIN) libraries
- Flists (field lists)
- Context (**PortalContext**)
- Buffers (**Buffer**)
- Fields (**Fields**)
- Portal object IDs (**POID**s)

For detailed information on BRM architecture and concepts, see:

- BRM System Architecture
- Understanding the PCM API
- Understanding the BRM Data Types
- Understanding Flists

Installation

PCM C++ is installed automatically as part of the BRM SDK. For installation and configuration information, including software requirements, see "About BRM SDK".

> **Note:**
>
> PCM C++ is sometimes be referred to as PCM CPP.

Comparison of the PCM C++ and PCM C APIs

Table 44-1 compares the functionality available in the PCM C++ and PCM C APIs:

Table 44-1 Comparison of C++ and C APIs in PCM

PCM C API	PCM C++ API
PCM_CONNECT	PinContextOwner PinContext::create( );
PCM_CONTEXT_OPEN	PinContextOwner PinContext::create(PinFlistBase & inFlist);
PCM_CONTEXT_CLOSE	void PinContext::close();
PCM_OP	N/A
PIN_FLIST_CONCAT	N/A
PIN_FLIST_COPY	PinFlistOwner PinFlist::clone() const;
PIN_FLIST_COUNT	N/A
pin_flist_sort	void PinFlist::sort(PinFlistBase &sortlist, int32 descending=0, int32 sortDefault=0);
pin_flist_recursive_sort	void PinFlist::sortRecursively(PinFlistBase &sortlist, int32 descending=0, int32 sortDefault=0);

PIN_FLIST_FLD_DROP	void PinFlist::drop(const PinXXXTypeField &fld);
	// For field types, where XXX is
	// Int, Uint, Num, Enum, Tstamp, Str, Binstr, Poid, BigDecimal

Table 44-1 (Cont.) Comparison of C++ and C APIs in PCM

PCM C API	PCM C++ API

PIN_FLIST_FLD_GET	PinXXXObserver
	PinFlist::get(const PinXXXTypeField &fld, PinBool optional=false);
	//Where XXX is
	//Int, Uint, Num, Enum, Tstamp, Str, Binstr, Poid, BigDecimal, Sub

PIN_FLIST_FLD_PUT	void PinFlist::set(const PinXXXTypeField &fld, PinXXXOwner &val);
	// For field types, where XXX is
	// Int, Uint, Num, Enum, Tstamp, Str, Binstr, Poid, BigDecimal, Sub

PIN_FLIST_FLD_SET	void PinFlist::set(const PinXXXTypeField &fld, PinXXX val);
	// Value based setter for simple field types, where XXX is
	// Int, Uint, Num, Enum, Tstamp, Str, Binstr
	void PinFlist::set(const PinXXXTypeField &fld, PinXXXBase &val);
	// For field types, where XXX is
	// Int, Uint, Num, Enum, Tstamp, Str, Binstr, Poid, BigDecimal, Sub

PIN_FLIST_FLD_TAKE	PinXXXOwner PinFlist::take(const PinXXXTypeField &fld, PinBool optional=false);
	// For field types, where XXX is
	// Int, Uint, Num, Enum, Tstamp, Str, Binstr, Poid, BigDecimal, Sub

PIN_FLIST_ELEM_ADD	PinFlistObserver PinFlist::add(const PinArrayTypeField &fld, PinRecId id)
PIN_FLIST_ELEM_COUNT	int PinFlist::count(const PinArrayTypeField& fld);
PIN_FLIST_ELEM_DROP	void PinFlist::drop(const PinArrayTypeField& fld, PinRecId id);
PIN_FLIST_ELEM_GET	PinFlistObserver PinFlist::get(const PinArrayTypeField& fld, PinRecId id, PinBool optional = PIN_BOOLEAN_FALSE);
PIN_FLIST_ELEM_GET_NEXT	PinElemObservingIterator::next();
PIN_FLIST_ELEM_PUT	void PinFlist::put(const PinArrayTypeField& fld, PinFlistOwner&, PinRecId id);
PIN_FLIST_ELEM_SET	void PinFlist::set(const PinArrayTypeField& fld, PinFlistBase&, PinRecId id);
PIN_FLIST_ELEM_TAKE	PinFlistOwner PinFlist::take(const PinArrayTypeField &fld, PinRecId id, PinBool optional = PIN_BOOLEAN_FALSE);
PIN_FLIST_ELEM_TAKE_NEXT	PinElemObservingIterator::next();
PIN_FLIST_SUBSTR_ADD	N/A
PIN_FLIST_SUBSTR_DROP	N/A
PIN_FLIST_SUBSTR_GET	N/A
PIN_FLIST_SUBSTR_PUT	N/A
PIN_FLIST_SUBSTR_SET	N/A
PIN_FLIST_SUBSTR_TAKE	N/A
PIN_FLIST_TO_STR	// String toString(); PinBool PinFlist::isNull() const;pin_flist_t* PinFlist::get();pin_flist_t* PinFlist::release();

Table 44-1 (Cont.) Comparison of C++ and C APIs in PCM

PCM C API	PCM C++ API

PIN_POID_COMPARE	int PinPoid::compare(const PinPoidBase &poid, int checkRev=0) const;
	//isEqual() returns a boolean
	//unlike compare() which acts like //strcmp();PinBool PinPoid::isEqual(const PinPoidBase &poid, int checkRev=0) const;

PIN_POID_COPY	PinPoidOwner PinPoid::clone() const;
PIN_POID_CREATE	static PinPoidOwner PinPoid::create(PinPoidDb db, PinPoidType type, PinPoidId id);
N/A	static PinPoidObserver PinPoid::createAsObserved(poid_t *pdp); static PinPoidOwner PinPoid::createAsOwned(poid_t *pdp);
PIN_POID_DESTROY	static void PinPoid::destroy(PinPoid *obj);
PIN_POID_GET_DB	PinPoidDb PinPoid::getDb() const;
PIN_POID_GET_ID	PinPoidId PinPoid::getId() const;
PIN_POID_GET_TYPE	PinPoidType PinPoid::getType() const;
PIN_POID_GET_REV	PinPoidRev PinPoid::getRev() const;
PIN_POID_IS_NULL	PinBool PinPoid::isNull() const;
PIN_POID_IS_TYPE_ONLY	PinBool PinPoid::isTypeOnly() const;
PIN_POID_TO_STR	void PinPoid::toString(char buf[ ], int bufsize, int skiprev=0) const;
pbo_decimal_abs	PinBigDecimal::abs() const;
pbo_decimal_abs_assign	N/A
pbo_decimal_add	PinBigDecimal::operator+(const PinBigDecimal& val); PinBigDecimal::operator+=(const PinBigDecimal& val);
pbo_decimal_add_assign	N/A
pbo_decimal_compare	int PinBigDecimal::compare(const PinBigDecimal& val) const; PinBool PinBigDecimal::isZero() const;PinBool PinBigDecimal::isLessThanZero() const;PinBool PinBigDecimal::isGreaterThanZero() const;
pbo_decimal_copy	N/A
pbo_decimal_divide	PinBigDecimal::operator/(const PinBigDecimal& val); PinBigDecimal::operator/=(const PinBigDecimal& val); //same result: PinBigDecimal::divide(const PinBigDecimal& val, int decimalPlaces, int mode = DEF_ROUNDING_MODE);
pbo_decimal_divide_assign	N/A
pbo_decimal_from_double	PinBigDecimal::setDouble(double val, int decimalPlaces, int mode = DEF_ROUNDING_MODE);
pbo_decimal_from_str	N/A
pbo_decimal_is_null	PinBool PinBigDecimal::isNull() const;
pbo_decimal_is_zero	See pbo_decimal_compare

Table 44-1 (Cont.) Comparison of C++ and C APIs in PCM

PCM C API	PCM C++ API
pbo_decimal_multiply	PinBigDecimal::operator*(const PinBigDecimal& val); PinBigDecimal::operator*=(const PinBigDecimal& val); //same result: PinBigDecimal::multiply(const PinBigDecimal& val, int decimalPlaces,int mode = DEF_ROUNDING_MODE);
pbo_decimal_multiply_assign	N/A
pbo_decimal_negate	PinBigDecimal::negate() const;
pbo_decimal_negate_assign	N/A
pbo_decimal_round	N/A
pbo_decimal_round_assign	N/A
pbo_decimal_signum	int PinBigDecimal::sigNum()
pbo_decimal_subtract	PinBigDecimal::operator-(const PinBigDecimal& val); PinBigDecimal::operator-=(const PinBigDecimal& val);
pbo_decimal_subtract_assign	N/A
pbo_decimal_to_double	PinBigDecimal::getDouble() const;
pbo_decimal_to_str	char* PinBigDecimal::toString(char* pbuf, int bufSize, int decimalPlaces = -1) const;

Understanding PCM C++ Concepts

While there are several similarities between the PCM C and PCM C++ APIs, there are also some key differences. This section explains these differences in detail and explains how to approach some specific programming functions. Understanding this information allows you to take advantage of PCM C++ capabilities and should make coding easier in areas such as memory management.

Passing Arguments

Using the C++ wrappers, you can minimize the need to handle untyped void pointers and explicit casting. In addition, PCM C++ includes methods that accept typed arguments. These two contrasting examples illustrate this point:

Using pointers in C:

```
time_t *endp = NULL;
time_t end_time;

endp = (time_t *) PIN_FLIST_FLD_GET(inflistp, PIN_FLDT_END_T, 1, ebufp);
end_time = (endp) ? *endp : pin_virtual_time(NULL);

PIN_FLIST_FLD_SET(outflistp, PIN_FLD_END_T, &end_time, ebufp);
```

Passing arguments in C++ :

```
PinTstampObserver endt   = inFlist->get(tsf_PIN_FLD_END_T, PIN_BOOLEAN_TRUE);

PinTstamp endTime = endt->isNull() ? pin_virtual_time(NULL):: endt->value();
```

```
outFlist->set(tsf_PIN_FLD_END_T, endTime);
```

In the C++ example:

- When retrieving *END_T* from the flist, no explicit casting is needed to convert from "*void**" to "*time_t**".
- When setting the value of *END_T* in the flist, you pass a typed argument (value) instead of an untyped pointer (pointer to the value).

Using Arrays

To walk through elements of an array in an flist, PCM C exposes a cookie-based interface, where it is the responsibility of the caller to initialize the cookie and pass it to a series of calls to retrieve the members of the array:

```
//Example in C initializting and passing a cookie
pin_cookie_t cookie = NULL;
while ((elemp = PIN_FLIST_ELEM_GET_NEXT
                (flistp, PIN_FLD_BALANCES, &elemid, 1, &cookie, ebufp)) != NULL) {
                ...
}
```

In contrast, PCM C++ uses iterator objects, which eliminate some common programming mistakes and also introduce commonly used patterns. A sample of iterator objects in C++:

```
//Example in C++ using iterator objects
PinElemObservingIterator iter;
for (iter = flist->getElements(tsf_PIN_FLD_BALANCES);
    iter.hasMore();) {

    PinFlistObserver aResource = iter.next();
    cout << "Resource id: " << iter.getRecId();
    ...
}
```

Using Smart Pointers to Manage Memory

C++ wrappers use constructs, generically called smart pointers, which are similar to the *auto_ptr* in the standard C++ library. Smart pointers are objects designed to look and act like built-in pointers, but offer greater functionality: smart pointers overload the operator -> (and sometimes the operator *).

PCM C++ uses smart pointers to eliminate the need for careful and explicit memory management that is required in PCM C. Because the smart pointer objects used to manipulate flists, POIDs, and connections are eliminated when the object is deleted or goes out of scope, the underlying resource, such as the flist, Portal object ID (POID), or connection, is freed.

> **Note:**
>
> Though it is common for smart pointers to be implemented using C++ templates, the PCM C++ implementation of smart pointers does not use them. This ensures consistent behavior on all BRM platforms.

Construction and Destruction

All flist, POID, and connection manipulations in PCM C++ are accomplished by using smart pointers. Smart pointers delete the object they point to when the last smart pointer pointing to that object is destroyed. This nearly eliminates resource leaks.

This example shows how a smart pointer is used to create an input flist:

```
void

function1()
{
      // Create input flist
      PinFlistOwner input = PinFlist::create();

      // Note that at the end of this block, the destructor
      // for "input" will get invoked. The destructor will
      // automatically destroy the underlying flist.
}
```

The smart pointer implementations in PCM C++ do not allow multiple smart pointer references to the same underlying object. Some of the reasons are:

- The underlying C implementation of flist does not allow reference counting, which adds overhead.
- A typical flist usage pattern requires a single pointer to an flist.
- Even done correctly, multiple pointers to the same flist can result in errors that are difficult to debug.

Copying and Assignment

In PCM C++, object ownership is transferred when the copy constructor or the assignment operator is invoked on a smart pointer. This is similar to the *auto_ptr* that is a standard C++ template library object. To copy the underlying data, use **clone()**. Because object ownership is transferred when the copy constructor is called, passing these smart pointers by value is not recommended.

The correct usage is to pass smart pointers using references, as shown below:

```
void function1()
{
    // Connect to Portal
    PinContextOwner context = PinContext::create();

    // Get the package list
    // (Note that the context smart pointer is passed by reference, not by value.
    // If it were passed by value, the underlying connection would have been destroyed
    // when function2() finishes and also "context" would be pointing to
    // a freed connection!!)
    function2(context);
}

void function2(
    PinContextBase &context)
{
    ...
}
```

Using Field Value Ownership

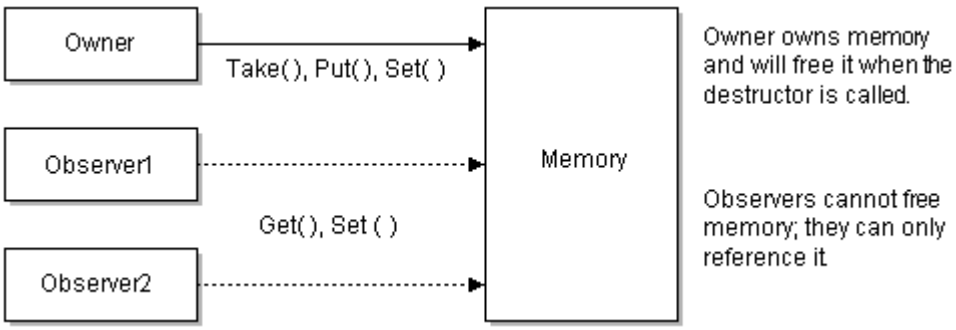
Manipulating flists is done differently in PCM C++ than in PCM C. The PCM C API provides different sets of functions to manipulate flists.

- PIN_FLIST_FLD_GET(), PIN_FLIST_SUBSTR_GET(), and PIN_FLIST_ELEM_GET() are used to access to the contents of the flist using pointers. This is like peeking inside the flist. The underlying flist retains ownership of the memory.
- With PIN_FLIST_XXX_TAKE(), the underlying flist relinquishes ownership of the pointed to block of memory and returns a pointer to the caller. The caller is now responsible for freeing the contents.
- PIN_FLIST_XXX_SET() is used to add new fields into an flist. The flist makes a copy of the passed in memory block and owns the copied block.
- PIN_FLIST_XXX_PUT() transfers ownership of field values to the flist.

PCM C++ provides overloaded methods in the **PinFlist** class to handle these four methods: **get()**, **take()**, **set()**, **put().** Correct memory management is enforced by defining two types of smart pointers: observers and owners as shown in Figure 44-1.

- Observers do not delete the wrapped BRM data structure (object) pointed to when they are destroyed.
- Owners delete the object pointed to when they are destroyed, unless they are assigned or passed to a copy constructor.

Figure 44-1 Owner and Observer Memory Management



The **PinFlist::get()** method returns an observer. The **take()** method returns an owner style smart pointer. Similarly, the **put()** method accepts only an owner. The **set()** method accepts either an observer or an owner. Relying on method signatures to do clean memory management is preferable to manual and careful programming. The following example shows the proper usage:

```
PinIntObserver flags = input->get(tsf_PIN_FLD_FLAGS);
// The following put() call will not compile because
// you cannot transfer ownership of something that you
// do not own!!
//      output->put(tsf_PIN_FLD_FLAGS, flags);

// The set() will work fine...because a copy is made.
output->set(tsf_PIN_FLD_FLAGS, flags);
```

Using PinBigDecimal

The PCM C++ **PinBigDecimal** class can be used with many standard C++ programming features. However, there are some differences which are documented in this section.

For a sample program illustrating PinBigDecimal, see **sample_PinBD.cpp** located in *BRM_SDK_home***/source/samples/apps/C++**.

Field Value Ownership

The PCM C++ class **PinBigDecimal** does not formally support the owner/observer model used by other PCM C++ classes. This alternative approach allows you to directly create and manipulate a **PinBigDecimal**, which makes it much easier to use in arithmetic expressions.

PinBigDecimal creates a base *object*, which is destroyed automatically when it goes out of scope. This concept is described in "Using Smart Pointers to Manage Memory".

This sample shows how to use **PinBigDecimal**. As noted above, PCM C++ creates a PinBigDecimal object that is destroyed automatically when it goes out of scope (in this example, when the function is exited).

```
{
     PinBigDecimal num1( "22.57" );
     ...
}
```

Using PinBigDecimal with Flists

PCM C+ uses a different approach than PCM C to setting (or getting) a *PinBigDecimal* value into (or from) an flist. In PCM C, you must carefully program using low-level functions: the PIN_FLIST_FLD_SET macro with the **PinBigDecimal.get()** method to retrieve the pointer to the actual big decimal number in *pin_decimal_t**. This is shown below:

```
PinBigDecimal sum = num3 + num4;

PinErrorBuf  ebuf;
pin_flist_t* flistp     = PIN_FLIST_CREATE( &ebuf );
PIN_FLIST_FLD_SET( flistp, PIN_FLD_DUE, sum.get(), &ebuf );
::printFlist( "The Flist I just created with C API...", flistp );
PinBigDecimal due( (pin_decimal_t*) PIN_FLIST_FLD_GET( flistp, PIN_FLD_DUE, 1, &ebuf ));
PIN_FLIST_DESTROY( flistp, 0 );
due.toString( buf, sizeof( buf ), due.getNumDecimalPlaces() );
::printf( "The value pulled from the Flist for PIN_FLD_DUE is: %s\n\n", buf );
```

In PCM C++, this low-level approach is unnecessary. Because C++ is object oriented and type safe, you can **set()** a **PinBigDecimal** *object* to the flist, which is shown below:

```
PinBigDecimal num3( 22.578979, 5 );
PinBigDecimal num4( double_val, 2, ROUND_HALF_DOWN );
PinBigDecimal sum = num3 + num4;

PinFlistOwner cpp_flist = PinFlist::create();
cpp_flist->set( tsf_PIN_FLD_DUE, sum );
::printFlist( "The Flist I just created with the C++ API...", cpp_flist->get() );
due = cpp_flist->get( tsf_PIN_FLD_DUE )->get();
due.toString( buf, sizeof( buf ), due.getNumDecimalPlaces() );
::printf( "The value pulled from the Flist for PIN_FLD_DUE is: %s\n\n", buf );
```

Using the toString() Method

To use PinBigDecimal with **toString()**, you must pass a parameter containing the number of decimal places to be set in the returned string. To do this, call **PinBigDecimal::getNumDecimalPlaces()**.

> **Note:**
>
> In some implementations, this is unnecessary because there is a default value for the number of decimal places. This is not yet implemented in PCM C++.

The following sample shows one method of converting a PinBigDecimal to a string by passing the necessary parameters.

```
char buf[100];
PinBigDecimal num1( "22.57" );
PinBigDecimal num2( "99"  );
PinBigDecimal sum = num1 + num2;

PinErrorBuf  ebuf;
pin_flist_t* flistp     = PIN_FLIST_CREATE( &ebuf );
PIN_FLIST_FLD_SET( flistp, PIN_FLD_DUE, sum.get(), &ebuf );
::printFlist( "The Flist I just created with C API...", flistp );

PinBigDecimal due( (pin_decimal_t*) PIN_FLIST_FLD_GET( flistp, PIN_FLD_DUE, 1, &ebuf ));
PIN_FLIST_DESTROY( flistp, 0 );
due.toString( buf, sizeof( buf ), due.getNumDecimalPlaces() );
::printf( "The value pulled from the Flist for PIN_FLD_DUE is: %s\n\n", buf );
```

Using the Divide Method

Because the decimal result can be infinite (for example 1/3=.333333...), the divide method of **PinBigDecimal** requires you to define the number of decimal places in the result. (The number of decimal places is called the scale.) Limiting the scale of the result introduces a rounding question, which is answered by defining the rounding method to be used. Defaults for both the scale and rounding method are defined in the **PinBigDecimal.h** file: the default scale is 10 [*DEF_DIV_DECIMAL_PLACES (10)*] and the default rounding method is "half up" [*DEF_ROUNDING_MODE (ROUND_HALF_UP)*].

This sample shows the divide method of **PinBigDecimal** using the default scale:

```
char buf[100];
char buf1[100];
char buf2[100];

PinBigDecimal num1 = "18.4328";
PinBigDecimal num2 = "3.4937";

PinBigDecimal num3 = num1 / num2;

num1.toString( buf, sizeof( buf ), num1.getNumDecimalPlaces() );
num2.toString( buf1, sizeof( buf1 ), num2.getNumDecimalPlaces() );
num3.toString( buf2, sizeof( buf2 ), num3.getNumDecimalPlaces() );

     cout << buf << " / " << buf1 << " is: " << buf2 << endl;
```

To programmatically specify the number of decimal places and/or the rounding mode, you must call the **divide** method, as shown in this example:

```
// The above statement "num3 = num1 / num2" will NOT be the same as the following call
// to the divide method.

PinBigDecimal num4 = num1;
num4.divide( num2, 9, ROUND_HALF_UP );
num4.toString( buf2, sizeof( buf2 ), num4.getNumDecimalPlaces() );

     cout << buf << " divided by " << buf1 << " is: " << buf2 << endl;
```

To programmatically specify the number of decimal places and/or the rounding mode, you must call the **divide** method.

Using a Null Pointer

You can use a special *null* pointer as a value to indicate that a feature is unused. For example, if a customer chooses not to enroll in an optional stock purchase plan, you can pass an flist with a null pointer for this option to the database to indicate that the customer is not participating.

Performing an arithmetic function on a **PinBigDecimal** variable containing a null pointer causes the **class PinBigDecimal** to throw an exception. However, you can use a null pointer successfully with the **try** and **catch** commands. This sample code illustrates this:

```
char* pnull = 0;
PinBigDecimal null_val = PinBigDecimal( pnull );
try
{
      PinBigDecimal bad_idea;  // The default constructor will set the value to zero.
      bad_idea += null_val;
      ::printf( "The program should NEVER get here...\n\n" );
}
catch ( const PinEbufExc& /*cExcptn*/ )
{
      null_val.toString( buf, sizeof( buf ), 2 );
      ::printf( "The string value of a null PinBigDecimal is: %s\n\n", buf );
}
```

> **Note:**
>
> `PinEbufExc` is the specific error thrown under these circumstances.

Handling Exceptions

PCM C is macro-based and uses series-style **ebuf**-based error checking. Since the same structure is used by all the function calls, all calls return immediately without any action after the first error is recorded in the **ebuf**. This style avoids an explicit error check after every line and allows you to group error handling logic toward the end of the function. The disadvantage is that almost every line following the error-inducing statement has the potential to be run.

In PCM C++, exceptions are used instead, which takes advantage of the support provided by the language run-time. The error handling in PCM C++ primarily uses an exception class,

PinEbufExc, which is a wrapper for the underlying C data structure (*pin_errbuf_t*). Use the class, **PinErrorBuf**, to access *pin_errbuf_t*.

There is one important difference: in the C API, when returning errors from a function using **ebuf**, you can return other values, such as output flists, by using output parameters. However, in C++, when an exception is thrown, the normal path of return is not available. To accommodate passing error flists back, the **PinEbufExc** class has a data member, **PinFlistOwner**.

This example shows error handling in PCM C++ using this method:

```
ostream&
operator<<(
ostream &os,
PinEbufExc &exc)
{
        os << "Pin Exception";
        os << exc.getFlistRef() << endl;
        PIN_LOG(exc, PIN_ERR_LEVEL_ERROR, "");
        return os;
};
```

This example shows error handling in PCM C++ using the exception buffer:

```
try {
      // Connect to Portal
      PinContextOwner context = PinContext::create();

} catch (const PinEbufExc &exc) {
      // Handle the error.
      PIN_LOG(exc, PIN_ERR_LEVEL_ERR, "Connect failed");
}
```

See **Pin_Log** for more information.

If an error occurs while processing an flist, the returned flist contains the error message. The following three code excerpts show how to print the flist to various devices.

This example shows the special overridden operator used to print out an flist:

```
ostream&
operator<<(ostream &os, PinFlistBase &flist)
{
      char *strp = NULL;
      PinErrorBuf ebuf;
      int32 len = 0;

      pin_flist_t *fp = NULL;
      if (! flist.isNullWrapperPtr()) {
            fp = flist->get();
      }
      // convert to string
      PIN_FLIST_TO_STR(fp, &strp, &len, &ebuf);
      // print out to current stream
      os << strp;

      if (strp != NULL) {
            pin_free(strp);
      }

      return os;
```

```
};
```

To print the flist to the console:

```
// Print output flist
cout<<"outFlist:"<<endl<<outFlist<<endl;
```

The **PinEbufExc** object contains the **PinErrorBuf** object, which is inherited from the PCM C structure, *pin_errbuf*. This buffer contains all the information about the error.

In PCM C++, define your own operator <<:

```
ostream&
operator<<(ostream &os, PinEbufExc &exc)
{
      os << "Pin Exception";
      os << exc.getFlistRef() << endl;
      PIN_LOG(exc, PIN_ERR_LEVEL_ERROR, "");
      return os;
};
```

and then use it to print **PinEbufExc** to the console and write information

to **PinLog**:

```
} catch (PinEbufExc &exc) {
      cout << exc << endl;
}
```

Logging to pinlog

The **PinLog** class is a minimalist class. It only provides type-safe wrappers that accept the PCM C++ class instances as arguments to the overloaded **log()** method.

Two macros, PIN_LOG and PIN_MSG**,** use the **PinLog** class. They allow you to pick up the current file and line number. Three examples of logging are:

```
PIN_LOG(flist, PIN_ERR_LEVEL_DEBUG, "Input to XXX");

PIN_LOG(ebufException, PIN_ERR_LEVEL_ERROR, "Create Account:");

PIN_MSG(PIN_ERR_LEVEL_WARNING, "Exceeding Cache Size");
```

For more information, see **Pin_Log** and **Pin_Msg**.

Accessing Configuration Values by Using pin.conf

The **PinConf** class provides static methods to get configuration values from a **pin.conf** file. Since the underlying **pin_conf()** PCM C library function returns an allocated memory block, the **PinConf** class type safe methods return owner-style smart pointers.

This example uses **PinConf** to access configuration values in a **pin.conf** file:

```
PinIntOwner dbg = PinConf::getInt("ldap_ds", "debug", 1);
int32 pinLdapDebug = (dbg->isNull() ? 0 : dbg->value());
```

Using PCM C++ with PCM C

There are many situations where you might want to mix the PCM C and PCM C++ APIs.

- Although PCM C++ provides useful abstractions, it is not complete. For example, it does not support buffer data types.
- New functionality based on PCM C++ might be developed and has to coexist with legacy code written in PCM C API. Also, this approach allows you to experiment and become familiar with PCM C++ without having to rewrite entire applications.
- Some PCM C API code is needed in rare situations, for example, invoking the PCM_OP_SEARCH opcode. PCM C++ enables the coexistence and mixing of the two APIs within the same application as follows:

```
PinFlistObserver flist = PinFlist::create();
// Get the underlying C flist data structure
pin_flist_t *flistp = flist->get();
// Pass the C data structure to some C function
PIN_FLIST_PRINT(flistp, 0, ebufp);
```

- This allows access to the underlying PCM C API objects that PCM C++ manages. For example, the **PinFlist** class can access the PCM C flist it holds. Obviously, doing destructive things to the underlying C object will make the C++ object inconsistent.

  Factory methods of the various PCM C++ classes take in pointers to PCM C API data structures, in addition to default factory methods that can create the PCM C API data structure automatically. Also, depending on the model of interfacing with the PCM C API, you can control the lifetime of the C data structures by creating observer or owner smart pointers, as in this example:

```
main() {
      pcm_context_t *ctxp = (pcm_context_t*) NULL;
      int64 dbno = 0;
      PCM_CONNECT(&ctxp, &dbno, ebufp);
      ...
}
PinContextObserver context = PinContext::createAsObserved(ctxp);
```

Using the PCM C++ API

The basic structure of a BRM PCM C++ client application is similar to other BRM client applications:

1. Open a connection using **PinContext**.
2. Create an flist.
3. Perform PCM operations.
4. Check for errors.
5. Close the connection.

Opening a PCM Connection

The **PinContext** class is a wrapper around the *pcm_context* data structure in the PCM C API. The data structure represents a connection from a BRM client to the server--a Connection Manager (CM).

Use the factory method **create()** to initiate a connection to the CM. This method uses the connection parameters from either an flist or as specified in the **pin.conf** file of the client application. Use either of these methods to open a connection:

- Call the **create()** factory method in the **PinContext** class. Using parameters contained in an flist, provide all the login information needed, including the host name and the port number. A program to be used by CSRs can use this method to force authentication.

  For a C++ sample, look for the **create_context.cpp** sample file in *BRM_home***/source/samples/context/C++** directory, where *BRM_home* is the directory in which the BRM server software is installed. For more information, see PCM_CONTEXT_OPEN.

- If you want your program to automatically log in to BRM, use the **create()** method without parameters. You must store all the necessary login information in the *cm_ptr* and the *userid* connection parameters in your application's **pin.conf** file. Use one **pin.conf** file to configure C++ and C applications. A billing application run from a cron job would use this method.

  For more information on the pin.conf file, see "Adding or Changing Login Options".

  For more information, see PCM_CONTEXT_OPEN and PCM_CONNECT.

> **Note:**
>
> In your application, when you open a context and connect to the BRM server, perform all the PCM operations before closing the context. Connections add a significant overhead to the system which affects performance. Therefore, to improve performance, perform all the operations within an open context instead of opening and closing contexts frequently. Use CM proxy for applications that cannot maintain an open context for a long time. For more information, see "Using CM Proxy to Allow Unauthenticated Log on" in *BRM System Administrator's Guide*.

Like the other PCM C++ classes, **PinContext** class instances are manipulated by using smart pointer classes: **PinContextOwner** and **PinContextObserver**.

This example shows a connection that gets the logon parameters from the application's **pin.conf** file:

```
try {
      // Connect to Portal. Get the connection info from the pin.conf file of the
      // client application.
      PinContextOwner context = PinContext::create();

      // The connection is terminated automatically and the PinContext
      // object managed by the PinContextOwner is destroyed automatically.

} catch (PinEbufException &exc) {
      // Fix the error.
      PIN_LOG(exc, PIN_ERR_LEVEL_ERR, "Connect failed");
}
```

When you create a context using either method (**PinContextOwner** or **PinContextObserver**), a *pcm_context_t* data structure is created automatically in C. A pointer to the data structure is automatically returned (access it by using **call Get()**). You can create additional context objects by using this pointer. Use this to pass a data structure to another application.

Closing a PCM Connection

Close a connection to BRM by using PCM C++ with one of the following methods:

- A context can be closed automatically by going out of scope. This occurs when smart pointers have been used. The context is automatically closed when the function ends.
- To close the connection context (opened as an Observer) with the server (and the data structure in *pcm_context_t*) before the function ends, use this method:

  ```
  void close(int how=n)
  ```

- To close the connection context (opened as an Owner) with the server and to destroy a **PinContext** object before the function ends, use this method:

  ```
  static void destroy(PinContext *obj)
  ```

> **Note:**
>
> If the context was opened as Owner, **destroy** also closes the connection context.

The parameters shown above are described in **PCM_Context** class.

For more information on **PinContext** functions, see **PinContext**.

Creating Custom Fields

To create custom fields using PCM C++:

1. In the **customfld.h** file, create your #define manually. For example, assume that a custom int field called `CUSTOM_FLD_AGE` is defined as follows:

   ```
   #define CUSTOM_FLD_AGE       10001
   ...
   #DEFINE CUSTOM_FLD_AGE PIN_MAKE_FLD(PIN_FLDT_INT, 10001)
   ```

   Table 44-2 lists the field ID ranges for Oracle-only use and customer use.

   Table 44-2 BRM Field ID Restrictions

Field ID Range	Reserved For
0 through 9999	Oracle use only
10,000 through 999,999	Customer use
1,000,000 through 9,999,999	Oracle use only
Over 10,000,000	Customer use

2. Instantiate a new C++ object:

   ```
   const PinIntTypeField tsf_CUSTOM_FLD_AGE(CUSTOM_FLD_AGE);
   ```

> **Note:**
>
> By convention, the prefix *tsf_* is used for the C++ version of the field. *tsf* is an acronym for type-safe field.

3. Include **customfld.h** in your application.
4. Use the custom field with the **PinFlist** class:

```
flist->set(tsf_CUSTOM_FLD_AGE, 22);
```

Creating an Flist

To create and use flists in PCM C++, use the flist factory method.

1. Create the input flist using **PinFlist::create()**.
2. **set** the values into the flist using the methods available in the **PinFlist** class. The suggested convention is to preface the variable name with "tsf_" (type safe field).

Several sample programs are available that illustrate:

- Simple flists.
- Flists with arrays.
- Flists with substructs.

For more information on the PCM C++ sample files, see "About Using the PCM C++ Sample Programs" in *BRM Developer's Reference*.

Getting an Flist in Text Format

When debugging, it is often useful to read a text representation of an flist. The **FList** API provides the **toString()** method for general purposes. See "Using the toString() Method".

Debugging PCM C++ Programs

Write a sample C++ program to test connections, load flists from files, use the flists as input when calling opcodes on the server, and to display the returned flist.

For more information on error handling, see "Handling Exceptions".

This sample uses overloaded << operators to print information about lists and POIDs to the console. This is helpful in debugging.

```
ostream&
operator<<(ostream &os, PinPoidBase &poid)
{
      char str[512];
      poid->toString(str, sizeof(str));
      os<<str;
      return os;
};
```

Troubleshooting

In PCM C++, object ownership is transferred when the copy constructor or the assignment operator is invoked on a smart pointer. This behavior can cause errors that are difficult to debug, as shown below:

```
class function1
{
      PinPoidOwner m_AcctPoid
      ...
};

...

function1::process()
{
    ;process()
     {
          PinPoidOwner t=m_AcctPoid;
         // The assignment owner took the memory. When the block ends, t goes
         // out of scope, the destructor is called, and the memory is freed.
         // An error might occur because m_AcctPoid now points to freed memory.
       ...
     }
     ...

// The following will crash because memory was freed when t went out of scope.

Print(m_AcctPoid);
```

Part VI

Customizing Customer Center and Self-Care Manager

This part describes how to customize Oracle Communications Billing and Revenue Management Customer Center and Self-Care Manager. It contains the following chapters:

- Using Customer Center SDK
- Customizing the Self-Care Manager Interface
- Customizing the Customer Center Interface
- Using Configurator to Configure Customer Center
- Adding Custom Fields to Customer Center
- Setting Up JBuilder to Customize the Customer Center Interface
- Creating a New Customer Center Service Panel
- Creating a New Customer Center Profile Panel
- Sample Customer Center Customizations

45

Using Customer Center SDK

Learn how to use the Oracle Communications Billing and Revenue Management (BRM) Customer Center Software Development Kit (SDK) to customize the Customer Center and Self-Care Manager client applications. While Customer Center and Self-Care Manager share many of the controllers for communicating with BRM, their client interfaces differ.

Topics in this document:

- About Customer Center SDK
- Contents of Customer Center SDK
- Coding Your Customizations
- About Compiling and Packaging Your Customizations
- Syntax for the buildAll Script
- Testing Your Customizations for Customer Center
- Understanding the BRM Business Application SDK Framework
- Source Code Examples

About Customer Center SDK

Customer Center SDK provides the framework and toolkit you need to customize and configure the default implementations of Customer Center and Self-Care Manager. To install Customer Center SDK, see *BRM Installation Guide*.

About Using Customer Center SDK to Customize Customer Center

Customer Center SDK allows you to modify the appearance and behavior of Customer Center to meet your business needs. You make these changes by using the customization APIs available for each screen of Customer Center.

Customer Center SDK includes a client application called Configurator that provides a graphical user interface for making modifications and additions to the default Customer Center properties and resources. Configurator enables you to do the following:

- Change Customer Center tab order or tab contents
- Remove certain Customer Center fields
- Change Customer Center behavior
- Change the fields displayed in account search results

For information on using Configurator to make these changes to Customer Center, see "Customizing the Customer Center Interface".

> **Note:**
>
> Because of interdependencies between fields that reside on multiple panels, certain fields cannot be removed from the Customer Center interface. Additionally, existing fields cannot be rearranged.

The SDK also includes scripts, customized properties files, source code examples, and a utility for exploring and copying field definitions that you can use for extending or customizing Customer Center functionality.

For customizing or modifying Customer Center Help, the SDK provides the online help in a compressed file. Scripts are provided for unpacking the file and creating a new custom help file for deployment from your Customer Center deployment servers.

For information on deploying Customer Center changes you make using Customer Center SDK, see "About Compiling and Packaging Your Customizations".

About Using Customer Center SDK to Customize Self-Care Manager

Self-Care Manager allows your customers to log into their accounts and view their account and product information by using a Web browser. Customer Center SDK allows you to modify the appearance and behavior of Self-Care Manager to meet your business needs.

Customer Center SDK includes the following components for customizing Self-Care Manager:

- HTML code you can edit to change the appearance of the web pages.
- Java Server pages (JSPs) for changing the layout of the Self-Care Manager interface, removing elements included in the default JSPs, or using a different set of components.
- Examples of modified JSPs and custom controllers for extending Self-Care Manager functionality.
- Scripts that build and package a custom Self-Care Manager Web Application Archive (WAR) file for deploying your customized files. See "Modifying the Self-Care Manager WAR File" in *BRM Managing Customers* for information about deploying an updated WAR file.

Contents of Customer Center SDK

Customer Center SDK includes these components.

For modifying Customer Center and Self-Care Manager:

- **buildall**, a batch file for building your customized Customer Center and Self-Care Manager code. See "About Compiling and Packaging Your Customizations".
- "Customer Care API Reference", a set of JavaDocs describing the Customer Center SDK classes.

For modifying Customer Center only:

- **Configurator**, a graphical utility for implementing many common GUI modifications
- **Scripts**, for testing your code.

Customer Care API Reference

See *Customer Care API reference* for JavaDocs describing Customer Care classes.

Customer Care API reference is also available in your *CCSDK_home***CustomerCareSDK\docs** directory. To view the documentation, use a Web browser to open the **index.html** file in this directory.

Coding Your Customizations

To design and code your customizations, see "Customizing the Customer Center Interface" or "Customizing the Self-Care Manager Interface".

About Compiling and Packaging Your Customizations

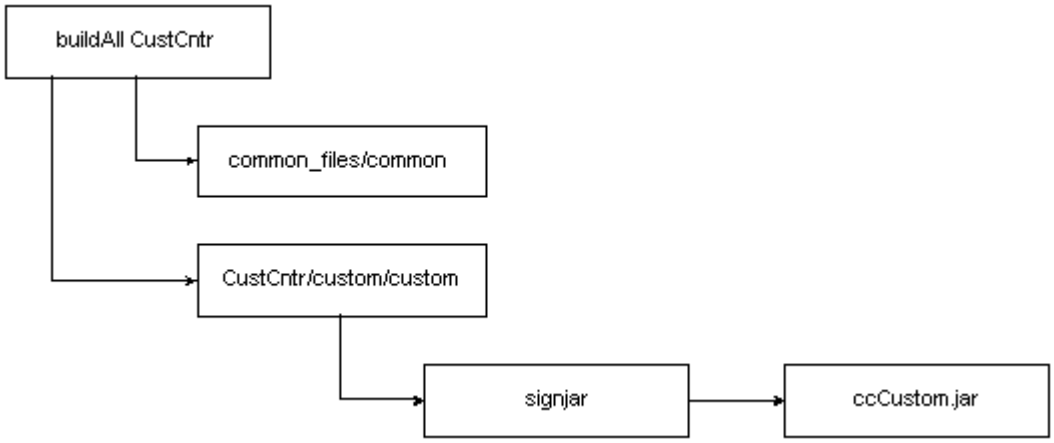
You compile your customizations to Customer Center and Self-Care Manager by using the **buildall** script in the *CCSDK_home***CustomerCareSDK** directory, where *CCSDK_home* is the directory in which the Customer Center SDK components are installed. This script calls other scripts to accomplish the following tasks:

- For Customer Center:
  - Compiles any Java source code files encountered in the *CCSDK_home***CustomerCareSDK\CustCntr\custom** directory.
  - Repackages the newly compiled class files and the **WizardCustomizations** and **Customized** properties files into a *CCSDK_home***CustomerCareSDK\CustCntr\custom\ccCustom.jar** file.
  - Signs the new **ccCustom.jar** file with a Java Security Certificate.
- For Self-Care Manager:
  - Compiles any Java source code files encountered in the subdirectories under the *CCSDK_home***CustomerCareSDK\WebKit\custom** directory.
  - Repackages the newly compiled class files into a **webkit_en.war** file.
  - Builds a new *CCSDK_home***CustomerCareSDK\WebKit\custom\webkit_en.war** file.

Figure 45-1 and Figure 45-2 show the relationships between the **buildall** script and the scripts it calls to accomplish these tasks:

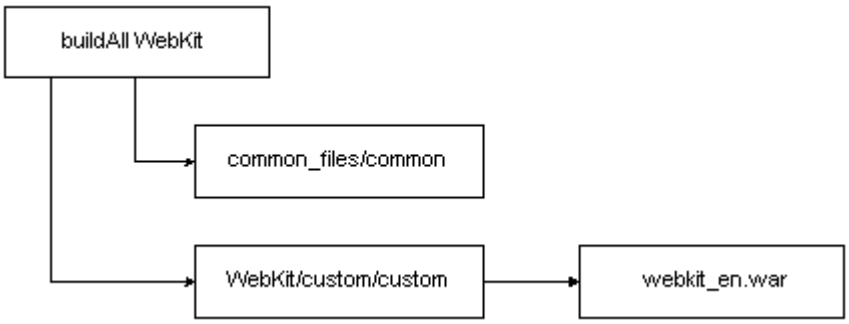
- For building a **ccCustom.jar** file for Customer Center:

Figure 45-1 Building Customer Center



- For building a **webkit_en.war** file for Self-Care Manager:

Figure 45-2 Building Self-Care Manager



This script calls the following scripts to process your files:

- **custom** compiles any **.java** files found in the **CustCntr\custom** or **WebKit\custom** directories.
- **signjar** (Customer Center only) signs the new **ccCustom.jar** distribution file with a security certificate, either a valid one obtained from an authorized provider or a self-signed certificate created with the **makecertificate** script. Before running the **buildAll** script, be sure to edit **signjar** to add values to the KEYPASSWORD and STOREPASSWORD entries.
- **common** compiles any**.java** files for controllers used by both Customer Center and Self-Care Manager found in the **common_files** directory.

> **Note:**
>
> Do not run the **custom**, **signjar**, or **common** scripts directly. They are designed to be called by **buildAll** only.

Coding, Building, and Deploying Customizations

To code, build, and deploy your customizations, see:

- For Customer Center, "Building Your Customer Center Customizations".
- For Self-Care Manager, "Building the Self-Care Manager Components".

Syntax for the buildAll Script

The **buildAll** script is used to build the customized **jar** files for your Customer Center and Self-Care Manager customizations.

This section describes the syntax for the **buildAll** script. For general information on how to use the **buildAll** script for the client applications, see:

- For Customer Center, "Building Your Customer Center Customizations"
- For Self-Care Manager, "Building the Self-Care Manager Components"

Syntax

```
buildAll CustCntr|WebKit [clean]
```

File Location

*CCSDK_home***CustomerCareSDK**

Parameters

Table 45-1 lists the parameters for the **buildAll** script.

Table 45-1 buildAll Script Parameters

Parameter	Description
CustCntr	Use this parameter to compile any new source code you put in the *CCSDK_home***CustomerCareSDK\common_files** and **CustCntr\custom** directories. The **ccCustoml.jar** file is created and signed.
WebKit	Use this parameter to compile any new source code you put in the *CCSDK_home***CustomerCareSDK\common_files** and **WebKit\custom** directories. The **webkit_en.war** file is created.
clean	Use the **clean** parameter with the **CustCntr** or **WebKit** parameters to verify that class files left from a previous build are removed before running **buildAll** again to rebuild the custom **jar** or **war** file. You must specify the **CustCntr** or **WebKit** parameters, not both, when you use the clean parameter.

Testing Your Customizations for Customer Center

To test the customizations you made to Customer Center:

1. Run the **runCustomerCenter** script, in the *CCSDK_home***CustomerCareSDK\CustCntr\bin** directory.
2. When the Login dialog box appears, enter your login information, and then click **OK**.

A local version of the Customer Center application starts. Any customizations you made are represented in this local version of Customer Center.

Understanding the BRM Business Application SDK Framework

Customer Center and Self-Care Manager use the Business Application SDK (BAS) framework. The BAS framework supports the construction of a set of distributed objects and mutually independent Portal Infranet-aware (PIA) components.

The Model-View-Controller Architecture

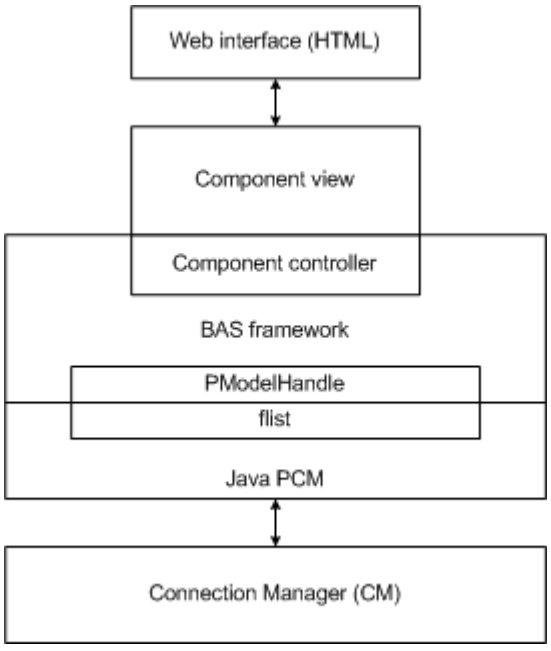
The BRM business applications framework is based on the model-view-controller (MVC) paradigm. In this paradigm, a component's control logic is separated from its display:

- The **model** component is the data. BRM data and BRM data types exist only in the controllers.
- The **view** component is the display of the data in the user interface. The view component handles only the Java data.
- The **controller** component is the logic on the server that determines the behavior of the data and how the data is displayed. The controllers convert the BRM data into Java data such as **Vector**, **int**, **Date**, and **String**.

In Customer Center and Self-Care Manager, the controller and view components are combined.

Figure 45-3 shows the Customer Care architecture:

Figure 45-3 Customer Care Architecture



How the Controllers Work

This section describes the basic controller process:

1. When users interact with a component, the component calls **setModelHandle**, which sets the appropriate property for the field and sends the model handle to the controller.
2. The component calls the controller's **update** method.
3. The controller communicates with BRM by calling the necessary opcodes.
4. The opcodes get the fields specified in the field specification or they get data after performing calculations or other functions, depending on the component.

Example Data Flow between a Simple Field and BRM

This example shows the data flow between the view component of a simple text field in the Account Maintenance panel, its controller, and BRM:

1. An external action, such as selecting an account in the Search Results panel, initiates a call to **setModelHandle** on a text field.
2. The text field view component calls its controller to get the data that corresponds to its **displayFieldDescription**.
3. The controller communicates with BRM by calling opcodes or reading objects.
4. The controller reads the **/account** object in the database and gets the fields specified by the display field specification.
5. The controller computes a new value and saves it in a property, for example, **mField=val**.
6. The view component calls a **get** method, for example, **getField()**, to get the previous property, and then sets this value in its display with **setText(getField())**.

About Field Components

Field components have model handles and the Customer Center SDK components contain references to field subclasses in the BRM database. These components have several properties, for example:

- **Model field description**

  Specifies the data that a field contains. For example, the first name field has this specification for model field: **FldNameInfo[1].FldFirstName**.

- **Display field description**

  Specifies the data to display. For example, the read-only field of an **account** object might have this specification for the display field: **FldNameInfo[1].FldFirstName,FldNameInfo[1].FldLastName, FldAccountNo**.

- **Display field format**

  Specifies the display format for the data specified in the display field. An **/account** object might have this specification for the display field format: **{0}{1} - {2}** to display the account information in the "First name Last name - Account number" format or **{1}{0} - {2}** to display "Last name First name - Account number" format. The field numbers, such as **{0}**, **{1}**, correspond to the order in which they are specified in the display field specification.

Every field component must have a specification for its **modelFieldDescription**, its **displayFieldDescription**, or both. The properties of the field depend on the type of the field.

For example, a read-only field does not have a **modelFieldDescription** property, but it does have **displayFieldDescription** and **displayFieldFormat** properties.

> **Note:**
>
> Field component names follow the Java class-naming conventions and use mixed case without the underscores. The Java field classes in the Java Portal Communication Module (PCM) package included in Customer Center SDK use the C **#define** name without the PIN_ prefix. For example, **PIN_FLD_NAME_INFO** in C becomes **FldNameInfo** in Java. See "Creating Client Applications by Using Java PCM" for information about Java PCM. See *Customer Care API reference* for the **com.portal.pcm.fields** package for information about the supported field subclasses.

Displaying Versus Saving Data in Fields

You use fields in a panel to display data as well as to allow users to change data. When users change data, their changes must be saved in the database. To implement reading from and writing to the object field in the database, you need to understand the read and write functions in BRM.

For information on BRM read and write fields functions, see "Base Opcodes".

For most fields that you create, you use the **modelFieldDescription** property to display and save changes to the data. For example, to display as well as save changes to the first name in the **FldNameinfo** field for the billing contact, you can use the following property for both **modelFieldDescription** and **displayFieldDescription**:

```
FldNameinfo[1].FldFirstName
```

However, not all fields have the same string for their model and display field descriptors.

In cases where reading the data is different from writing the data, such as the **FldPayinfo** data:

- For reading the data, set the description in **displayFieldDescription**.
- For writing the data, set the description in **modelFieldDescription**.

For example, use the following **displayFieldDescription** to read the credit card number field:

```
FldPayinfoObj.FldccInfo[0].FldDebitNum
```

And use the following **modelFieldDescription** to write the credit card number field:

```
FldPayinfo[1].FldInheritedInfo[0].FldccInfo[0].FldDebitNum
```

See *Customer Care API reference* for **PCreditCardPanel.java** for examples of setting the description in this manner.

Portal Infranet-aware (PIA) components:

- Are Swing compatible.
- Have references to specific storable class fields.
- Contain APIs that allow you to map them to specific fields in BRM storable classes, such as **/account**.

- Have both client and server implementations, which allow individual components to encapsulate data retrieval and user interface (UI) display.

Display Fields and Controllers

You can use properties, such as a **modelFieldDescription** and a **displayFieldDescription**, when there is a one-to-one correspondence between the fields in the Customer Center interface and the fields in the BRM object. For example, fields such as first name and last name have fields of type **String** in the **/account** object that map exactly to the display. Therefore, you can specify their data by specifying values for the **modelFieldDescription** and **displayFieldDescription** properties in a graphical IDE, such as JBuilder.

To display fields that do not have one-to-one mapping in the object, you need to extend an existing controller or create a new one. For example, the currency field is an integer type in the **/account** object. To display it as a string such as "US Dollar" in the user interface, you need to write a controller that:

- Takes the integer.
- Looks it up in the balance element ID (BEID) table.
- Gets the appropriate string description or name of the currency represented by the integer.

You can write controllers that perform any number of functions. For example, a controller can:

- Perform simple tasks, such as simple translations. For example, you can map an integer type to a string description and send it to the view component.
- Call opcodes and process information.
  - For example, to display the available payment methods, you can write a controller that calls the appropriate opcode, retrieves the data in a vector, and sends the vector to the view component.
  - You can initialize data (flists) for payment information.

> **Note:**
>
> * If you create a new Infranet-aware field, you must create a controller for the field.
> * (Self-Care Manager only) If you add a field component to a JSP that has a controller, you might need to add code to the JSP controller for any special processing of the new field. This code is necessary, for example, when you add custom fields to BRM and use Self-Care Manager to access them. For example, **com.portal.app.comp.PIAPhoneTableBeanImpl** handles the special processing for the phone fields for account creation. If you create a new Infranet-aware field, you need to create a controller for the field.

About PModelHandle

A **PModelHandle** is an object that passes a reference to a BRM data object. Because the view code does not have access to the PCM library, the BAS API creates this object for client applications to use.

The data referenced by a**PModelHandle** is converted from its specified **modelFieldDescription** to an flist or a storable class, such as **/account**, for the BRM opcodes to use.

When data is returned from BRM, the BAS API converts it into the format specified by the **displayFieldDescription** for that object and the results are displayed in the associated Customer Center field.

About Lightweight Components (Self-Care Manager Only)

Lightweight components such as text field components are only view components. They are used for storing and displaying data. They do not contain control logic. The encapsulating parent component or the page controller handles the lightweight components. They do not each need a controller.

Use lightweight components to map a UI field to a specific field in a BRM storable class, such as **/account**. The lightweight component used in Self-Care Manager is **PLightComponentHelper**. It is a wrapper for a field component (**PFieldBean**), which allows data to be retrieved from and passed to BRM.

For example, the credit card number UI field does not have a controller because it does not need information from the database; the "Choose service" UI field needs a controller to call the opcode to retrieve the list of available plans. So the credit card number is created as a lightweight component, but "Choose service" is not.

Source Code Examples

Customer Center SDK provides scripts and applications for customizing Customer Center and Self-Care Manager. When you use Configurator or make entries or changes to the editable properties files, the **buildAll** script creates a new custom distribution file you copy to the appropriate directory to deploy your changes in a production environment. For Customer Center, the script creates a new **jar** file signed with a digital certificate. For Self-Care Manager, the script creates a new **war** file.

Source code examples are provided with Customer Center SDK to show how to use some of the more useful public API available to you.

For more information, see "Customer Center Customization Examples" and "Self-Care Manager Customization Examples".

46

Customizing the Self-Care Manager Interface

Learn how to customize the Self-Care Manager interface.

Topics in this document:

- About Customizing Self-Care Manager
- Understanding Self-Care Manager Components
- Extending the Functionality of Self-Care Manager
- Self-Care Manager Customization Examples

See also:

- About Customizing BRM
- Understanding the PCM API

About Customizing Self-Care Manager

You can customize Self-Care Manager:

- By editing the JSPs to use a different set of components or by adding new JSPs.
- By creating new components and modifying existing components to extend the functionality of Self-Care Manager.

See "Extending the Functionality of Self-Care Manager".

> **Note:**
>
> Self-Care Manager files must be used for reference only. When you customize Self-Care Manager, you must implement security that prevents unauthorized access to the BRM system.

Hardware and Software

You require the following hardware and software to develop and test components for Self-Care Manager:

- Customer Center SDK

  For information on installing Customer Center SDK, see *BRM Installation Guide*.
- Java Development Kit.
- An application server that supports Servlet API and JSP.

Understanding Self-Care Manager Components

Self-Care Manager consists of:

- JSPs and servlets, which act as view components.
- Business logic (BAS) beans that act as controller components.
- The BAS framework, which handles connections to BRM and caches BRM data accessed by multiple components. See "Understanding the BRM Business Application SDK Framework".

About PInfranetServlet

PInfranetServlet is the main servlet which initializes BAS. It is a generic servlet that loads **WebKit.properties** and sets **PPooledConnectionClientServices** as the BAS client service. The BAS Connection Pooling Service tracks the connections in the pool. (These connections are shared by all Self-Care Manager users.) It also sets the start directories and the start page based on its configuration with the servlet engine.

PInfranetServlet servlet:

- Sets up a new connection to Connection Manager (CM) for new sessions by performing an explicit login on the BAS client service based on the properties defined in the **WebKit.properties** file. The connection is saved and used throughout the session and is only removed when the session is invalidated.

  Most components that the user interacts with use this connection. The exceptions are **CreateFormPage1.jsp** and **CreateFormPage2.jsp**. These components set up their own connections.

> **Note:**
>
> **The CreateFormPage1.jsp** and **CreateFormPage2.jsp.** These jsps are used to create an account and account creation is not supported in this release. It continues to be supported in the previous releases.

- Handles the user login component, validates the user, and saves this user's account model handle.

Using PInfranetServlet to Process Requests

To process requests using **PInfranetServlet**, you might need to specify the following in the HTML/JSP pages:

1. Specify **"page=<page_to_be_loaded_next>"** to forward the request to the page after creating the controller.
2. Specify **Component=<Class>** to:
   - Call BAS to create the controller.
   - Call the setters to set input data.
   - Call the public methods with either **HttpSession** or **PModelHandle** and **ResourceBundle** as arguments.

> **Note:**
>
> The **ResourceBundle** is a deprecated parameter and should *not* be used in the business logic of the controller.

3. (Optional) Specify **sessionstate=start** to create a new session:
   - It creates a data structure to be passed to BAS with the character set and package list information. The character set is based on the browser and is used to display invoices. The package list is specified in **WebKit.properties**.
   - It calls **registerApp** with the program name, locale, and the data structure created in the earlier step.
4. (Optional) Specify **sessionState=end** to invalidate the session.

> **Note:**
>
> You can use either **sessionState=start** or **sessionState=end**.

5. (Optional) Specify **validateBean=<Class>** to create the class and call the setters and the validate method for validation before calling the controller. If the validation fails, a **RemoteException** is thrown; otherwise, it sets the input data of the controller and calls the public methods. For example, specify the following in **invoice_selection.jsp** to use **DateValidator** to validate the user entered dates:

   <INPUT Type="hidden" Name="validateBean"
   Value="com.portal.web.fmt.DateValidator">

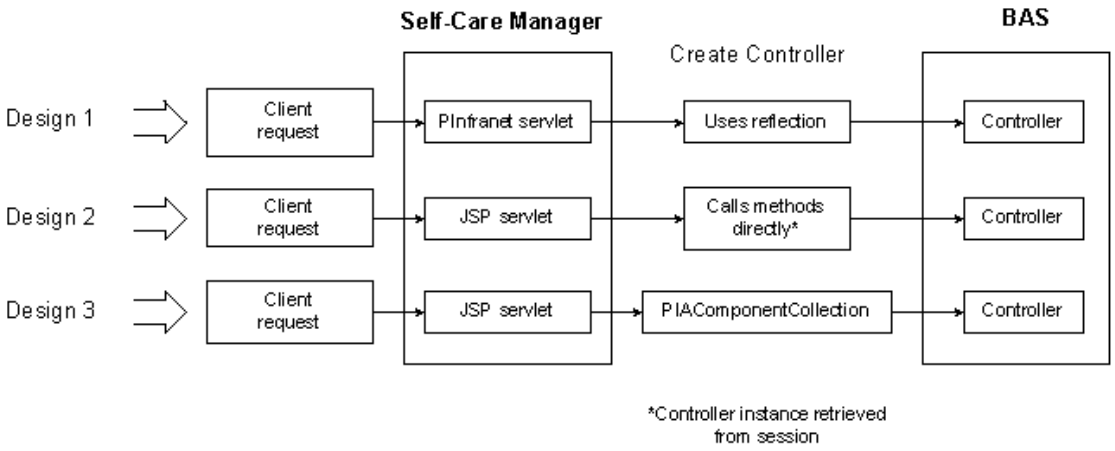
Example Data Flow Designs

This section describes three example designs for communication between the servlet and controller that you can implement in your Self-Care Manager customizations. Information on how to choose the appropriate design from the samples is described in "Designing a Component".

> **Note:**
>
> These designs are examples. You can implement other designs according to your business needs.

Figure 46-1 summarizes the three example design architectures:

Figure 46-1 Example Design Architectures Summary



1. When a client requests information, an HTTP request is sent to the application server and is processed by a servlet engine.
2. The servlet engine sends a request to the appropriate JSP servlet as specified in the request.

   For example, this request is sent to the **change_acct_form** JSP servlet:

   ```
   <FORM Action="change_acct_form.jsp" Method="post">
   ```

   This request is sent to **PInfranetServlet**:

   ```
   response.encodeURL ("PInfranetServlet?
   page=change_status_form&Component=com.portal.web.comp.PServicesBeanImpl&loadBean=yes"
   )
   ```

   > **Note:**
   >
   > Do not include the servlet path in the request.

   The HTTP request arrives at the application server and is dispatched to the appropriate servlet. Then one of three possible designs is followed:

Design 1

In Design 1, communication between the servlet and the controller occurs using introspection:

1. The request is received by the main servlet, **PInfranetServlet**.

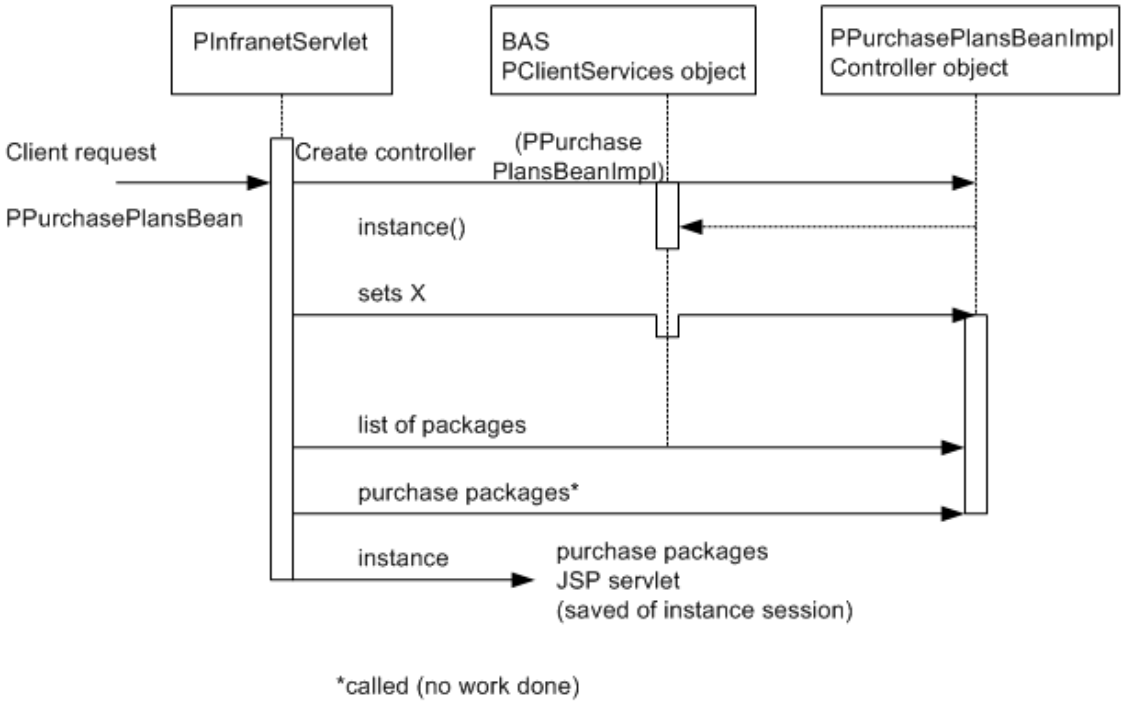
   This is a generic servlet that performs some common Self-Care Manager tasks.
2. **PInfranetServlet** collects all the input data and creates the BAS controller beans specified in the "component" hidden variable.
3. The servlet uses introspection to call all the setters to set the input data properties for this bean (if any) and to call all public methods.
4. The JSP servlet then forwards this bean instance to the JSP specified in the page hidden variable.

5. The JSP servlet loads the data from the bean and returns the HTML output to the browser.

   In some cases, this bean is saved in the session for use in later requests.

This Design 1 example UML sequence diagram in Figure 46-2 shows the data flow for displaying the packages available for purchase:

Figure 46-2 Design 1 Example



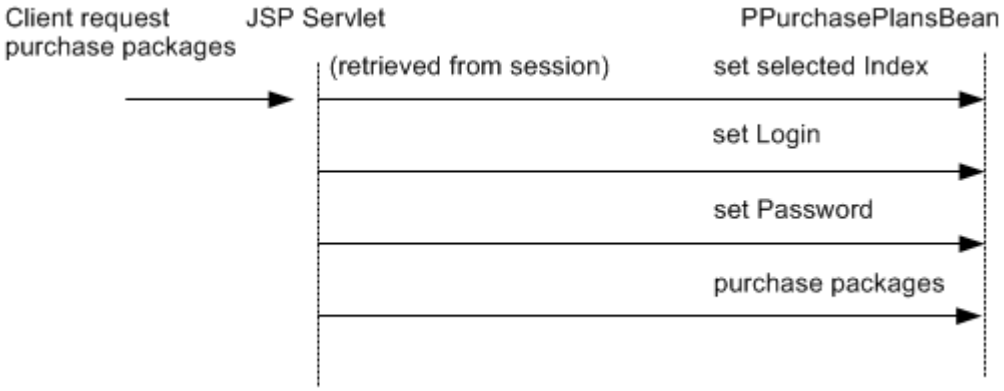
Design 2

In Design 2, communication between the servlet and the controller occurs by calling methods directly:

1. The request is received by a specific JSP servlet.
   - If the controller bean associated with this JSP servlet does not exist, one is created to perform the necessary operation(s).
   - If the bean exists, the servlet uses the bean saved in session.
2. The servlet:
   a. Collects all the input data.
   b. Uses the bean saved in the session to set the input data properties of this bean.
   c. Calls the methods directly to perform the necessary operation for data.
   d. Returns the HTML output to the browser.

This Design 2 example UML sequence diagram in Figure 46-3 shows the data flow for purchasing the package that the user selected:

Figure 46-3 Design 2 Example



Design 3

In Design 3, communication between the servlet and the controller occurs through **PIAComponentCollection**:

1. The request is received by a specific JSP servlet.

   In this design, the bean is not the controller, but is a component collection bean, that is, an instance of **PIAComponentCollection**.

2. The **createController()** method is called on the component collection bean to specify the controller bean that should be delegated to perform the necessary operation.

3. The lightweight components are added to the collection to specify the fields that should be retrieved from the database.

   For more information on lightweight components, see "About Lightweight Components (Self-Care Manager Only)".

4. It calls **setModelHandle()** on the component collection bean instance to retrieve the data of the model from the database.

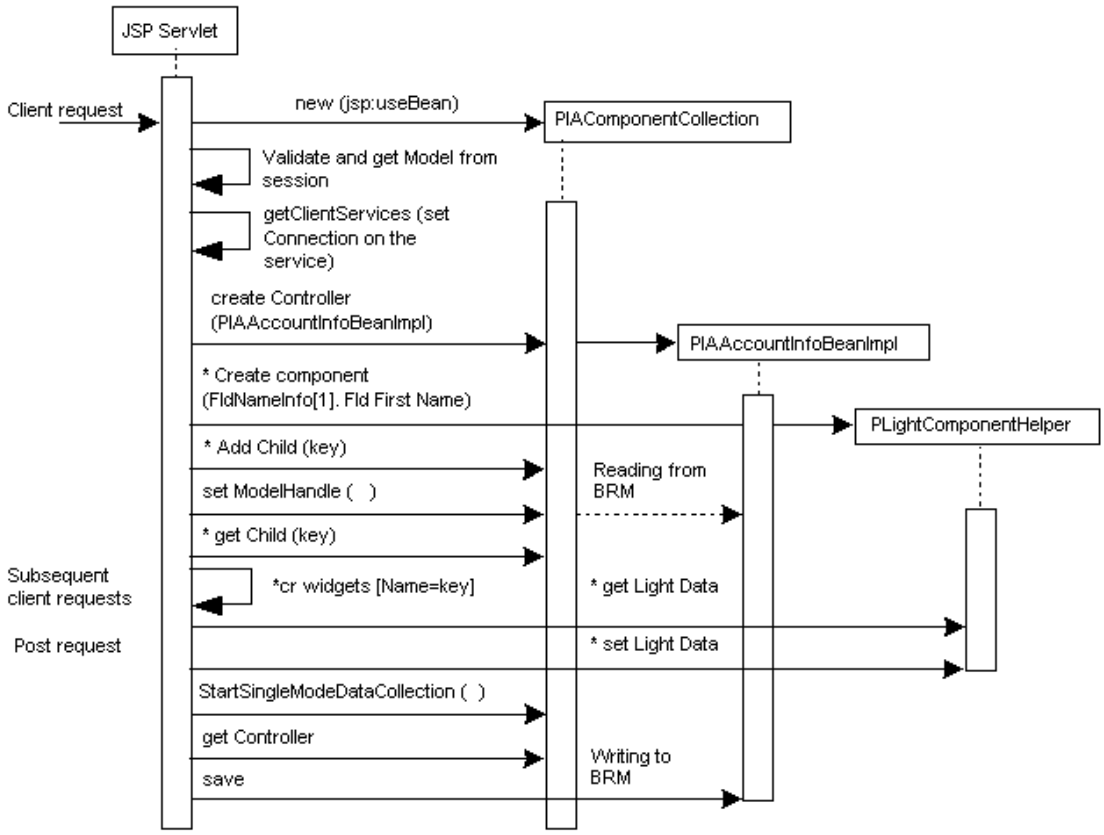
   Data can be displayed using the **getLightData()** method.

5. This JSP servlet collects any data needed and calls the **setLightData()** method on each of the lightweight components to set the input data properties.

6. To save data to the database, the servlet calls the methods of the controller bean directly to perform the necessary operation and then returns the HTML output to the browser.

This Design 3 example UML sequence diagram in Figure 46-4 shows the data flow for displaying account information. It is an example of a customizable JSP, where you can add or remove new account fields to be displayed to the user:

Figure 46-4 Design 3 Example



Extending the Functionality of Self-Care Manager

All Self-Care Manager components follow the model view controller (MVC) paradigm. A component in Self-Care Manager includes the controller, which contains the logic of the component, and the view, which handles the display of data on the UI. The view in Self-Care Manager is HTML/JSP. When you create a component for a Customer Care application by using the SDK, you need to create the view of the data and the controller that determines the behavior and display of the data.

Self-Care Manager includes a set of components that provide the basic functionality for customer self-care. All the properties that the default beans support are included in the JSPs.

You can extend the functionality of Self-Care Manager to collect additional information from customer accounts or to provide customers with additional options. For more information, see:

- Adding Fields
- Removing Fields
- Creating a New Component

> **Note:**
>
> The controllers used in Self-Care Manager have getters and setters; that is, they are true beans.

Adding Fields

You add fields by adding to the component collection using the **ServletUtil.addComponent(…)**, which is a wrapper for the BAS API to add a lightweight component to the collection. In addition to the collection bean object, this method requires the name of this lightweight component, the model field description, and the display field description for updating and retrieving the data to and from BRM.

The name of the lightweight component is the same as specified for the UI field; that is, for **firstname** if the UI field specification is:

<INPUT Name="firstname" Value="<%=fname%>" Size="21" Tabindex="2">

Then the name passed to **addComponent** is the value of **NAME**, that is, **firstname**.

The model field description is **FldNameinfo[1].FldFirstName.** In this case, the display field description is the same **"FldNameinfo[1].FldFirstName"**. However, for the credit card number, the model field description is **FldPayinfo[1].FldInheritedInfo[0].FldCcInfo[0].FldDebitNum** and the display field description is **FldPayinfoObj.FldCcInfo[0].FldDebitNum**.

> **Note:**
>
> When the collection is saved in the session, you can retrieve the lightweight components from the collection by using **getChild()**.

1. Retrieve the lightweight component data by using **getLightData()**.
2. When a customer enters data, **ServletUtil.gatherFormInput()** retrieves the user data. It then calls **setLightData()** of **PLightComponentHelper** to update the data of the component.

   > **Note:**
   >
   > **ServletUtil.setLightDataForAll()** is a wrapper function that calls **PLightComponentHelper.setLightData()** for every input field that has a lightweight component in the collection. However, if the input data needs massaging or there is no mapping between the UI field and the lightweight component, then you need to explicitly call **setLightData()**.

3. The return flists are parsed by calling **ServletUtil.parseErrorData**, which identifies lightweight components that are in error. You can mark the lightweight components that are in error by checking if the field is in the error flist. To check the error flist, call **ServletUtil.checkError(…)**.

Removing Fields

To remove a field that is not required, for example, the middle name field, you remove all references to it.

> **Note:**
>
> Before you remove a field, make sure that no opcode requires it by checking opcode input flist specifications.

1. Remove the **ServletUtil.addComponent(…)** of that field so it is no longer added to the collection.

   For example, to remove the middle name field, delete:

   ```
   ServletUtil.addComponent(<BEAN>, MIDDLENAME,
   "FldNameinfo[1].FldMiddleName"); and <BEAN>.
   getChild(MIDDLENAME).
   ```

2. Remove the calls to **checkError** and **getLightData()**.

   For the middle name field, remove the following:

   ```
   <% if (ServletUtil.checkError(errorMap, cbMname)) { %>
       <TH Align="right" Class="optional">__MiddleName__*</TH>
   <% } else { %>
       <TH Align="right" Class="optional">__MiddleName__</TH>
   <% } %>
       <TD  Colspan="3" Align="left"><INPUT Name="middlename"
   Value="<%=cbMname.getLightData()%>" Size="21" Tabindex="3"></TD>2.
   ```

3. If **setLightData()** is called explicitly on this lightweight component, remove that line.

Creating a New Component

When adding a component to Self-Care Manager, you create the view component of the data, and if necessary, the controller component that determines the behavior of the data.

1. Create a link in the existing JSPs or HTML pages for loading the next or new JSPs.

   See "Creating a Link for the JSP Pages for a Get Request".

2. Create the component.

   For a view component, subclass either an existing controller from **app/ccare/comp** or **web/comp,** or design a new one.

   See "Designing a Component".

3. Develop the customizable component.

   See "Developing the Customizable Component".

4. Develop the noncustomizable component.

   See "Developing a Noncustomizable Component".

Creating a Link for the JSP Pages for a Get Request

If you are using **PInfranetServlet** and the HTTP request is a get request, add the *page_to_load_nex*t and *component_to_be_created* values in the **response.encodeURL** parameter entry in the appropriate JSP:

```
A HREF="<%=response.encodeURL("PInfranetServlet?page=
       page_to_load_next&Component=component_to_be_created")%>"
```

For example:

```
A HREF="<%=response.encodeURL("PInfranetServlet?page=
       change_login_form&Component=com.portal.web.comp.PServicesBeanImpl")%>"
```

Creating a Link for the JSP Pages for a Post Request

If you are using **PInfranetServlet** and the HTTP request is a post request, add the following entry, specify the page to load next, the Component to be created and a submit button:

```
<FORM Action="PInfranetServlet"
       Method="post"><INPUT Type="hidden" Name="page"
       Value=page_to_load_next>
       <INPUT Type="hidden" Name="Component" Value=Component_to_be created>
```

For example:

```
<FORM Action="PInfranetServlet"
       Method="post"><INPUT Type="hidden" Name="page" Value="view_invoice"><INPUT
       Type="hidden" Name="Component" Value="com.portal.web.comp.PInvoiceBeanImpl">
```

If you are *not* using **PInfranetServlet**, use the same link you would for HTML pages.

Designing a Component

To design a component:

1. Plan the UI and functionality.
2. Determine if the component UI involves updating and displaying data.
   a. If the UI displays the data, then Design 1 or Design 3 is appropriate. See Choosing Design 3 to determine if you need to use it; if not, use Design 1.
   b. If the UI displays and updates data, use one of the following:

      - A combination of Design 1 to retrieve data and Design 2 to update it.

      - Design 3. To determine if you need to use Design 3, see Choosing Design 3; if you do not, use a combination of Design 1 and Design 2.
3. Determine the design that the data flow of your component resembles:
   a. See Choosing Design 3, to determine if you need to use it to build a customizable component.
   b. Choose Design 1 if you want most of the work to be done by **PInfranetServlet**; otherwise, choose Design 2.
   c. To share the same controller bean instance between multiple HTTP requests, choose the combination of Design 1 to retrieve data and Design 2 to update data.

Choosing Design 1 and Design 2

You can use Design 1 and Design 2 to display data and update it on user interaction. For example, you can display all the packages available for purchase. When the user selects a package, the controller bean updates the user information. Use Design 1 to display all the packages. The bean is saved in session. When the user selects a package, the same bean that has the model handle to the data is called with the index of the package. When the **save** method is called, the flist of the package at the selected index is retrieved from the model.

Choosing Design 3

Use Design 3 when you have a one-to-one correspondence between the fields in the view and fields in the object. Each of these fields can be a lightweight component. For example, fields such as **First Name** and **Last Name** in the **View Balance** page have object fields that map exactly to the display, so you have a light component for first name and last name. To display **FldBalances**, there is a specific controller that you specify by using the **createController()** method in **PIAComponentCollection**. This allows you to override the **update** method, which takes an integer reason code and object as parameters to do special processing, such as retrieving the balance information.

> **Note:**
>
> When you override an update you have to call **super.update()** so that any **FieldBeans** that are part of the collection also get updated; that is, their **update** method is called, so the opcode to retrieve data is run.

Developing the Customizable Component

This component has a view component and one or more controllers. For example, **CreateFormPage2.jsp** servlet has a few controllers.

> **Note:**
>
> To extend Self-Care Manager functionality, you must subclass the controllers provided in Customer Center SDK.

Developing the View Component

The view component displays the data and provides user interaction with the system.

To develop the view component, create a new HTML page and then follow these steps to edit it to build the JSP.

1. Add a page directive that sets the value for **errorPage** to **error.jsp**:

   ```
   <%@ page errorPage="error.jsp" %>
   ```

   This redirects all exception handling to the defined page. For information about handling exceptions, see "Error Handling".

2. Add a **jsp:useBean** statement, with the following values shown for **scope** and **class**:

```
<jsp:useBean id="myBeanInstanceName" type="InterfaceName" class="concrete
Implementation of the interface" scope="request"/>
```

> **Note:**
>
> You are saving the collection bean instance in session, not the data.

3. Follow the steps in Account creation or Account maintenance:
   - If you're adding functionality that is independent of an individual user logging in, such as account creation, follow the steps in Account creation.
   - If you're building functionality that is dependent on the user logging in, for example, account maintenance, follow the steps in Account maintenance.

> **Note:**
>
> Account creation and maintenance is not supported in this release. It continues to be supported in the previous releases.

Account Creation

> **Note:**
>
> This is an example for developing a customizable component. Account creation is not supported in this release.

Follow these steps for account creation:

1. If a client service exists for the session, retrieve it:

```
PPooledConnectionClientServices pCS = (PPooledConnectionClient)session.getAttribute
(CREATE_CONNECTION);
```

2. For a first-time access, that is, the collection bean was just created and the controller was not set:
   a. Create a new instance of **PPooledConnectionClientServices**:

```
pCS = new PPooledConnectionClientServices((PClientServices)
application.getAttribute(ServletUtil.PARENT_SERVICE));
>
```

   b. Call **setServices** with the new instance of **PPooledConnectionClientServices** on the collection bean:

```
accountCreationBean.setServices(pCS);
>
```

   c. Create a **ConnectionListener** and save it in the session:

```
ConnectionListener listener = new
ConnectionListener(session.getCreationTime(), pCS);
```

d. Call **ServletUtil.saveLocaleInfo** and **registerApp**.

e. Specify the controller associated with this component.

f. Create the lightweight components. See "Creating a New Component".

3. If the collection bean was created by a previous HTTP request, retrieve the lightweight components in the collection by calling **getChild** on the collection bean saved in the session.

4. Call **getLightData** on the lightweight component to display the data.

5. If the data can be updated by the user, use a POST request to handle the update. In the JSP code, check for POST requests, as follows:

   a. Call **ServletUtil.gatherFormInput**.

   b. To set data, call **ServletUtil.setLightDataForAll**.

      This function loops through the components. If there is a mapping between the component and user input, **setLightDataForAll** calls **setLightData** on the component.

   c. For user input for which where there is no mapping, such as a billing address, you can explicitly call **setLightData** on the component.

   d. After setting the user input values on all the components in the collection, collect the data for storing:

      ```
      accountInfoBean.startSingleModelDataCollection
           (PCollectDataEvent.FOR_STORING, <model>);
      ```

      The value for *model* is either null or the model handle previously created with some data. In the case of payment method, for example, create an untyped model but modify it with the payment info flist.

6. Call **session.invalidate** to release the connection.

Account Maintenance

> **Note:**
>
> This is an example for developing a customizable component. Account maintenance is not supported in this release.

Follow these steps for account maintenance:

1. Retrieve the model handle.

   ```
   PModelHandle mH = ServletUtil.getModelFromSession(session);
   ```

2. Use **ServletUtil.CONNECTION** as the key to retrieve the **PPooledConnectionClientServices** instance from the session.

> **Note:**
>
> A **PPooledConnectionClientServices** instance is created and saved in a session using **ServletUtil.CONNECTION** when a user logs in. Additional functionality provided for the user, such as account maintenance, can use **ServletUtil.CONNECTION** to retrieve the **PPooledConnectionClientServices** instance from the session.

3. Call **setServices** with the retrieved instance on the collection bean:

   ```
   accountInfoBean.setServices(pCS);
   ```

4. If this is a first-time access; that is, the collection bean was just created and the controller was not set:
   a. Specify the controller associated with this component.
   b. Create the lightweight components. See "Creating a New Component".
   c. Call **setModelHandle** to retrieve data from the database.
5. If the collection bean was created in a previous HTTP request, you can retrieve the lightweight components in the collection by calling **getChild** on the collection bean saved in the session.
6. Call **getLightData** on the lightweight component to display the data.
7. If data can be updated by the user, use a POST request to handle the update. In the JSP code, check for POST requests:
   a. Call **ServletUtil.gatherFormInput**.
   b. To set data, call **ServletUtil.setLightDataForAll**.

      This function loops through the components. If there is a mapping between the component and user input, it calls **setLightData** on the component.
   c. For user input where there is no mapping, such as a billing address, you can explicitly call **setLightData** on the component.
   d. After setting the user input values on all the components in the collection, collect the data for storing:

      ```
      accountInfoBean.startSingleModelDataCollection
           (PCollectDataEvent.FOR_STORING, <model>);
      ```

      The value for *model* is either null or the model handle previously created with some data. In the case of payment method, for example, create an untyped model but modify it with the payment info flist.

Developing the Controller Component

You must create a controller for each view element. The controller component performs the functions, for example, reading fields from the database.

To create the component controller:

1. Declare the controller's API in the interface:
   a. Declare all the setters to set input data.
   b. Declare all the getters to retrieve the output data.
   c. Declare all the public methods.

2. Define a class derived from **com.portal.bas.comp.PIAComponentCollectionBean**, which implements the interface.

   Implement all the public methods, setters and getters.

Developing a Noncustomizable Component

This component has a view component and one controller.

Developing a View Component

The view component displays the data and provides user interaction with the system.

To develop the view component, create the HTML page and then build the JSP as follows:

1. Add the page declarative which has the errorpage set to **error.jsp**.

   This redirects all exception handling to this page.

2. Add the **jsp:useBean** statement whose type is the controller interface and scope is request or session.

   If the bean was already created through the main servlet and placed in the request object, you can load the bean using the **jsp:useBean** clause and use the getters to get the data.

3. Call the getters to get any data.

4. If data can be updated by the user, it requires special handling. Use a POST request to handle the update. So in the JSP code, check if POST.

   a. Gets the model from the session, if it exists.

   b. Calls **ServletUtil.gatherFormInput**.

   c. Calls the setters of the light components to update them with user inputs.

   d. Calls the method to save the data into the database.

For information about error handling, see "Error Handling".

Developing the Controller Component

You must create a controller for each view element. The controller component performs the functions, for example, reading fields from the database.

To create the component controller:

1. Declare the controller's API in the interface, for example, in **PCAAccountInfoBean**:

   a. Declare all the setters to set input data.

   b. Declare all the getters to retrieve the output data.

   c. Declare all the public methods.

2. Define a class derived from **com.portal.bas.PControllerImpl**, which implements the interface defined in the previous step. Note these rules:

   - Implement all the public methods, setters and getters.
   - Use the connection pool to connect to BRM.
   - For error handling, see "Error Handling".

Using the Connection Pool

When developing a noncustomizable component, you must use connections from the connection pool to connect to BRM.

For more information on connection pooling, see "About PInfranetServlet".

> **Note:**
>
> The Self-Care Manager Connection pool implementation is expected to change in a future release of BRM.

When using the connection pool, note these rules:

- Use **getConnection** to get a connection from the connection pool.
- Use **releaseConnection** to release the connection back to the connection pool.

  > **Note:**
  >
  > – You must use **releaseConnection** to release connections.
  >
  > – You must pair a **getConnection** with a **releaseConnection**.

- Adjust the values of the **infranet.bas.connectionpool.size** and **infranet.bas.connectionpool.timeout parameters** in the *Self-Care_Manager_install_dir***/WebKit.properties** file.

  > **Tip:**
  >
  > If you add noncustomizable components to your Self-Care Manager implementation, Self-Care Manager performance may improve if you increase the value for the **infranet.bas.connectionpool.size** parameter from the default value **4**.

  For more information on these parameters, see "Optimizing Self-Care Manager Connection Pool Performance" in *BRM Managing Customers*.

Error Handling

The controller handles exceptions and errors shown in Table 46-1:

Table 46-1 Exceptions, Errors, and Responses

Exception/Error	Response
EbufException	Calls **PControllerImpl.createClientException()** in the BAS API
Other exception	Throws **RemoteException** with the resource string

Table 46-1 (Cont.) Exceptions, Errors, and Responses

Exception/Error	Response
Error returned in flist	Returns **CustomerValErrorData**

If the exception is received in **PInfranetServlet**, it redirects to **error.jsp**

If the exception is received by a JSP Servlet with the error attribute in **<page >** directive, set to **error.jsp**, which redirects the exception to **error.jsp**.

If you return **CustomerValErrorData**, call **ServletUtil.parseErrorData(...)** to gather the lightweight components in error, and call **ServletUtil.checkError(...)** to mark any data in error.

Formatting Your Data

You can customize the display format of your data by using one of these methods:

- Method 1: Add Java Code to Your JSP Pages
- Method 2: Use a Formatting Bean that Contains the Presentation Logic for the Data.

Method 1: Add Java Code to Your JSP Pages

You can specify the date in month-day-year format by using the following Java code in a JSP page:

```
<%Date lastBill = accountBean.getLastBillT();%>
<%=lastBill.getMonth()%>/<%=lastBill.getDate()%>/ <%=lastBill.getYear()%>
```

See the **view_balance.jsp** file included with Customer Center SDK in *CCSDK_home*/**CustomerCareSDK/WebKit/htmliu_en**.

You can edit the entry to change the format to day-month-year by changing the order:

```
<%Date lastBill = accountBean.getLastBillT();%>
<%=lastBill.getDate()%>/<%=lastBill.getMonth()%>/ <%=lastBill.getYear()%>
```

Method 2: Use a Formatting Bean that Contains the Presentation Logic for the Data.

The JSP calls the formatting bean to obtain the instructions on how to display the data. By using a formatting bean, you reduce the amount of Java code in the JSP.

To use a bean to format data:

1. Create a Java class with setters and getters to set the data and get the formatted data.
2. Compile the file and save the class file in a directory included in the CLASSPATH.
3. Add entries in your JSP page to do the following:
   - Point to the formatting bean.
   - Set the property of the data.
   - Get the property of the formatted data from the formatting bean.

This example shows the entries for formatting the date in the **view_balance.jsp**.

1. Define a class, such as **PSampleFmt**, with a **setDate()** and **getFormattedDate()** method. The **getFormattedDate()** method retrieves the date set and returns the formatted date.

2. In the **view_balance.jsp** file, replace the Java code for formatting the date with these lines:

```
<jsp:useBean id="fmtBean"class="com.portal.web.fmt.PSampleFmt"
scope="request"/>
<jsp:setProperty name="fmtBean" property="date"
value=<%=accountBean.getLastBillT()%>
</jsp:useBean>
<jsp:getProperty name="fmtBean" property="formattedDate"/>
```

You can use PPartialListFmt (in **com/portal/web/fmt**) to break down the display of events list into multiple pages with a NEXT button.

Building the Self-Care Manager Components

To build or rebuild the **webkit_en.war** file for Self-Care Manager, follow these steps:

1. If you created custom source to extend Self-Care Manager, copy it to the *CCSDK_home/***CustomerCareSDK/WebKit/custom** directory.
2. Copy any HTML or JSP pages you modified for the HTML version of Self-Care Manager to *CCSDK_home*/**CustomerCareSDK/WebKit/htmlui_en**.
3. Make the appropriate entries in the **WebKit.properties** file as required to accompany your source.
4. Open a command shell and enter the **buildAll** command with the appropriate syntax for the application you're customizing. See "Syntax for the buildAll Script".

   For example, to clean and rebuild the **webkit_en.war** file for Self-Care Manager, enter the following:

```
buildAll WebKit clean
buildAll WebKit
```

> **Note:**
>
> For more information on the **buildAll** script, see "About Compiling and Packaging Your Customizations".

Self-Care Manager Customization Examples

Customer Center SDK includes Self-Care Manager customization example code in *CCSDK_home***/CustomerCareSDK/WebKitExamples**.

When you install Customer Center SDK, source and support files provide examples for extending Self-Care Manager. Many of the example folders include **Readme.txt** files that explain the purpose of each example.

You can run many of the examples in place by using the **testExamples.bat** script.

Table 46-2 describes the Self-Care Manager extension examples in the *SDK_home***/CustomerCenterSDK/WebKitExamples** directory. Read the **readme.txt** files and the comments in the source files of each example for further information on their functionality and how to use them for creating your own customizations.

Table 46-2 Self-Care Manager Extension Examples

Directory Under *SDK_homel* CustomerCareSDK/WebKitExamples	Contents
Controllers	**PIAWKCreateAccountBeanImpl.java**, an example of subclassing **PIACreateAccountBeanImpl** to override its **validatePage** and **validate** methods.
Currency	**CreateFormPage1.java** and **CreateFormPage2.java**and examples that demonstrate how to include the currency field as input for account creation. **Note:** Account creation is not supported in this release.
Profile	**CreateFormPage1.jsp** and **CreateFormPage2.jsp**, examples that demonstrate how to add support for a profile object to Self-Care Manager. The profile object used for this example is **/profile/customertype**.

47

Customizing the Customer Center Interface

Learn how to customize the Oracle Communications Billing and Revenue Management (BRM) Customer Center interface.

Topics in this document:

- Customizing and Configuring Customer Center
- Modifying the Customer Center Properties Files
- Using Customer Center SDK Scripts for Customer Center
- Adding New Pages to the Customer Center Interface
- Advanced Customer Center Concepts
- Building Your Customer Center Customizations
- Deploying Your Customer Center Customizations
- About the Customer Center Properties Files
- Customer Center Customization Examples

For information about Java classes used in Customer Center SDK, see the *Customer Care API Reference*.

> **Note:**
>
> Customer Center Software Development Kit (SDK) uses the Portal Communication Module (PCM) Java Application Programming Interface (API) to communicate with BRM database objects. While the PCM API uses the format PIN_FLD_*SUBCLASS* to represent an flist field to BRM, the PCM Java API uses the format Fld*SubClass* to represent an flist to BRM. See the *API Reference for PCM Java* for information about the Java equivalents for the PCM API field subclasses.

Related documentation:

- To customize Self-Care Manager, see "Customizing the Self-Care Manager Interface".
- To use the Java PCM API to create Java client applications that interface with BRM, see "Creating Client Applications by Using Java PCM".
- To customize localized versions of Customer Center using Customer Center SDK and the Localization SDK, see "Modifying Localized Versions of Customer Center".

Customizing and Configuring Customer Center

This section describes the "Tools and Techniques for Customizing Customer Center" and a "Customization Procedure Overview" describing which tools and techniques to use for various types of customizations.

Tools and Techniques for Customizing Customer Center

To customize and configure the Customer Center interface, perform the following tasks:

1. Code your customizations by:
   - Using the graphical Configurator application to configure the most commonly modified interface features. See "Using Configurator to Configure Customer Center".
   - Modifying Customer Center customized properties files. See "Modifying the Customer Center Properties Files".
   - Creating custom fields and panels with PIA (Portal Infranet Aware) widgets by using an IDE tool such as JBuilder. See "Setting Up JBuilder to Customize the Customer Center Interface".
   - Using scripts, such as the one that starts a local version of Customer Center, for testing your customizations. See "Using Customer Center SDK Scripts for Customer Center".
2. Build and test your customizations by using scripts such as:
   - **buildAll** to build your **jar** files. See "Building Your Customer Center Customizations".
   - **runCustomerCenter** to launch a local version of Customer Center to test your customizations before deployment. See "Testing Your Customizations".
3. Deploy your customizations. See "Deploying Your Customer Center Customizations".

Customization Procedure Overview

This section describes which "Tools and Techniques for Customizing Customer Center" to use for various types of customizations.

Coding Your Customizations

- If you are making basic changes to the Customer Center interface, such as changing page field attributes, page order, or search fields, you use Configurator.

  For a complete description of the customizations you can make with Configurator and how to make them, see "Using Configurator to Configure Customer Center".
- If you are making minor customizations not handled by Configurator, such as changing the list of services available in the search panel, edit the Customer Center customized properties files.

  See "Modifying the Customer Center Properties Files".
- If you are adding new fields to Customer Center, use:
  - Developer Center
  - JBuilder
  - Configurator

  See "Setting Up JBuilder to Customize the Customer Center Interface" and "Adding Custom Fields to Customer Center".
- If you are adding a new profile or service panel, you use JBuilder and Configurator. See:
  - Setting Up JBuilder to Customize the Customer Center Interface
  - Adding New Pages to the Customer Center Interface

  - Creating a New Customer Center Profile Panel
  - Creating a New Customer Center Service Panel
  - The sample profile and service panel code in the **/Profile** and **/Service** directories in the *CCSDK_home***/CustomerCareSDK/CustCntrExamples** directory.
- If you are adding new account maintenance tabs or pages to the new accounts wizard, see "Adding New Pages to the Customer Center Interface".

  See also sample code in the *CCSDK_home***/CustomerCareSDK/CustCntrExamples** directory.
- If you are doing advanced customizations, see "Understanding the BRM Business Application SDK Framework" and "Advanced Customer Center Concepts".

Building and Deploying Your Customizations

All customizations are built and deployed using the same procedures no matter which tools and techniques are used to do the customizations. See "Building Your Customer Center Customizations" and "Deploying Your Customer Center Customizations".

Modifying the Customer Center Properties Files

This section describes the properties files used to define Customer Center behavior and how to modify those behaviors.

About the Default Customer Center properties Files

Default Customer Center settings are defined with parameter-value pairs in the following files in the *CCSDK_home***/CustomerCareSDK/CustCntr/Settings** directory:

- **CustomerCenter.properties**
- **CustomerCenterResources.properties**

> **Caution:**
>
> Do not delete or modify the default properties files.

For information on the parameters in the default properties files, see the comments in the files.

Modifying Behaviors Defined by the Default Properties Files

To change the behaviors in the default properties files:

- Add or modify parameters in the **Customized.properties** and **CustomizedResources.properties** files, which are located in the *CCSDK_home***/CustomerCareSDK/CustCntr/custom** directory. Parameters and values specified in the **Customized*.properties** files take precedence over values for identical parameters in the respective default properties files.

  For examples of changing Customer Center behavior by modifying the **Customized.properties** file, see:

  - Displaying Event Timestamps with Seconds Precision
  - Adding Inactive Product Status Indicators

  - Changing the List of Services Available in the Search Panel
  - Improving Account Search Performance
  - Changing Number Searches for GSM Services
  - Modifying the Shortcut Key Sequences
  - Specifying the Number of Bills Displayed in the Balances Tab
  - Suppressing the "Missing Login/ID" Message for Custom Service Panels
  - Changing the Maximum Number of Security Code Characters
  - Updating Notes Before Saving
  - Reminding CSRs to Customize Deals Before Completing a Purchase
  - Identifying Services by Device ID Rather than Login ID
  - Adding a Tax Exemption Type
  - Customizing Event Searches
  - Customizing Balance Group Searches
  - Customizing Product/Discount Searches
  - Customizing Service Searches
  - Hiding the Password Fields in Customer Center
  - Disabling the Child Amounts Check Box
- Add or modify parameters in the **WizardCustomizations.properties** and **WizardCustomizationsResources.properties** files, which are located in the *CCSDK_home***/CustomerCareSDK/CustCntr/bin** directory. Parameters and values specified in the **WizardCustomizations*.properties** files:
  - Supersede values for identical parameters in the respective default properties files.
  - Are *superseded by* values for identical parameters in the respective **Customized*.properties** files.

> **Note:**
>
> Values in the **WizardCustomizations*.properties** files are also modified by the Configurator utility.

To modify one of these properties files:

1. Open the file with a text editor.
2. Add appropriate parameter-value pairs.
3. Save the file.

After you modify the properties files and make any other customizations, see "Building Your Customer Center Customizations" and "Deploying Your Customer Center Customizations".

Displaying Event Timestamps with Seconds Precision

By default, Customer Center displays event timestamps with hours and minutes precision, such as, 4:30 p.m. You can configure Customer Center to display event timestamps with

seconds precision, such as 4:30:55 p.m., by enabling the **customercenter.datetime.showseconds** entry in the **Customized.properties** file.

To display event timestamps with seconds precision:

1. Open the *CCSDK_home*/**CustCntr/custom/Customized.properties** file in a text editor.
2. Add the following line after the comment statements:

```
customercenter.datetime.showseconds=true
```

3. Save and close the file.
4. Restart Customer Center.

Adding Inactive Product Status Indicators

To add inactive product status indicators:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. After the comment statements, add the following lines:

```
product.details.status.flags.waiting=Waiting for installation
product.details.status.flags.network=Network configured
product.details.status.flags.maintenance=Maintenance
```

3. Save your changes.

Changing the List of Services Available in the Search Panel

To change the list of services available in the Search panel:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. Add the following line after the comment statements, changing the value to match the total number of service types to display in the Search panel:

```
searchpanel.service.num=3
```

3. Add entries for each service type to display, for example:

```
searchpanel.service.type.0=admin_client
searchpanel.service.type.1=broadband
searchpanel.service.type.2=email
```

> **Note:**
>
> If you add a new service type to BRM, add a **searchpanel.service.type** entry for it in the **Customized.properties** file and be sure to increment the value for **searchpanel.service.num** accordingly.

4. Save your changes.

Improving Account Search Performance

You can improve Customer Center account search performance by reducing the maximum number of search results displayed. The default is **1000**.

To reduce the maximum number of search results displayed:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. Add the following line after the comment statements:

   ```
   search.accountsresults.displaylimit=<new value>
   ```

3. Save your changes.

Changing Number Searches for GSM Services

You search for telephone numbers when assigning numbers for a GSM service in Customer Center. Numbers are assigned when customizing services in the New Account or Purchase wizards or when changing the number for an existing service.

One of the search criteria on the Search Number dialog box is **Status**. By default, the **Status** list has these options-**<Not Specified>**, **New**, and **Unassigned**. If you choose **<Not Specified>**, Customer Center searches for numbers with a status of either **New** or **Unassigned**. It does not search for numbers with a status of **Assigned** or **Quarantined**.

If you have customized the list to include other status options, **<Not Specified>** also searches for the custom options.

You can change this default so that Customer Center adds **Assigned** and **Quarantined** to the **Status** list and searches numbers with **Assigned** or **Quarantined** status when you choose **<Not Specified>**:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. After the comment statements, add the following line:

   ```
   device.num.search.entry.panel.status.availability=false
   ```

3. Save the file.

By default, **device.num.search.entry.panel.status.availability** is set to **true**. When you set this property to **false**, the following takes place:

- The default **Status** list will also display **Assigned** and **Quarantined**.
- When you choose **<Not Specified>**, Customer Center searches for numbers of any status.

Modifying the Shortcut Key Sequences

Customer Center provides shortcut key sequences for many of the actions in the UI. You can customize the mnemonics by adding an updated property statement for each custom shortcut in the **Customized.properties** file.

> **Note:**
>
> Make sure you do not assign the same mnemonic to different shortcuts.

To customize the shortcut key sequences:

1. Open the **CustomerCenter.properties** file in the *CCSDK_home*/**CustCntr/Settings** directory.
2. Copy the line containing the mnemonic property you want to change. For example:

   ```
   search.startsearch.mnemonic=S
   ```

3. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
4. Paste in the line you copied.
5. Change the value assigned to the property, for example:

   ```
   search.startsearch.mnemonic=x
   ```

6. Save your changes.

Specifying the Number of Bills Displayed in the Balances Tab

In Customer Center, you can view detailed information about the bills associated with an account in the **Bills** section of the **Balances** tab. By default, the **Balances** tab displays a maximum of six bills for each account. You can customize the maximum number of bills displayed by adding a property to the **Customized.properties** file.

To customize the maximum number of bills Customer Center displays on the **Balances** tab:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. After the comment statements, add the following line:

   ```
   balance.default.bills.count =Count
   ```

   Replace *Count* with the number of bills you want Customer Center to display for each account on the **Balances** tab.
3. Save your changes.

Suppressing the "Missing Login/ID" Message for Custom Service Panels

You can create custom service panels that do not request users to enter their login IDs and passwords. However, when users exit the custom service panel, Customer Center generates a "Missing Login/ID. Do you want to fix the error now?" message.

To suppress the login error message in custom service panels, perform the following tasks:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. Set the **extended.***ServiceType***.required** entry to **false**:

   ```
   extended.ServiceType.required=false
   ```

3. Save and close the file.

Changing the Maximum Number of Security Code Characters

When you create an account in Customer Center, you can enter security codes on the **General** tab of the Account Creation wizard.

You can also modify security codes in the **Account Summary** section of the **Summary** tab.

The BRM database is set up to store a maximum of 30 characters for a security code. You can change the number of allowed characters by modifying the database and adding a property to the **Customized.properties** file.

To change the maximum number of security code characters:

1. In the Storable Class Editor, change the length of the PIN_FLD_ACCESS_CODE1 and PIN_FLD_ACCESS_CODE2 fields in the **/account** storable class.
2. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.

3. After the comment statements, add the following line:

   ```
   summary.securitycode.length=length
   ```

   Replace *length* with the number of characters you want to allow. The number cannot exceed the size of the PIN_FLD_ACCESS_CODE1 and PIN_FLD_ACCESS_CODE2 fields in the ACCOUNT_T database table.

4. Save the file.

Updating Notes Before Saving

By default, when you open an account in Customer Center, existing notes are cached. If two or more users have an account open at the same time and add notes, only one user's notes will be saved.

To prevent this from happening, you can add a property that directs Customer Center to refresh notes from the database before saving an account. This can slow the performance, but it ensures that notes entered by different users to the same account are not overwritten.

To have Customer Center refresh notes from the database:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. After the comment statements, add the following line:

   ```
   notes.management.option=refresh
   ```

3. Save the file.

Reminding CSRs to Customize Deals Before Completing a Purchase

If you create deals with a **Deal Customization** setting of **Required**, you will probably want to ensure that CSRs go to the Customer Center **Customize Product** page to offer your customizations. Customer Center automatically reminds CSRs to customize deals with a **Required** deal customization setting when they first select the deal for purchase. You can also remind CSRs to visit the **Customize Product** page before *completing* the sale by using the **customize.deal.enforce** setting.

You turn this option on and off by changing the **customize.deal.enforce** setting in the **Customized.properties** file.

This example turns this option on:

```
customize.deal.enforce = true
```

The default is **false**.

Identifying Services by Device ID Rather than Login ID

By default, Customer Center identifies a service owned by an account by using the service login ID, allowing you to find email or IP services owned by the account. You can configure Customer Center to identify services by using device IDs rather than login IDs by using a **Customized.properties** entry. You might do this, for example, if you offer mostly telco services.

To configure Customer Center to identify services by using device IDs rather than login IDs:

1. Open the **Customized.properties** file in the *CCSDK_home*/**CustCntr/custom** directory.
2. After the comment statements, add the following line:

```
service.alias.display=true
```

3. Save and close the file.

Adding a Tax Exemption Type

If a tax exemption type does not exist, you can include it by overriding an existing property in the **WizardCustomizationsResources.Properties** file:

1. Make a copy of the **WizardCustomizationsResources.Properties** file.
2. Add the following property:

```
exemptionType.format={0,choice,0#Federal|1#State|2#County|3#City|
4#Secondary County|5#Secondary City|8#District}
```

3. Place the customized file in the directory where you start Customer Center.

Customizing Event Searches

Customer Center has the following criteria for event searches for wholesale customers. Customer Center allows you to:

- Narrow your filters for event searches by providing search criteria in the Event Search dialog box. For more information, see Customer Center Help.
- Customize the case sensitivity of the searches for events. See "Customizing the Case Sensitivity of Event Searches".
- Customize event searches by adding custom settings for the following search components:
  - Service types. See "Customizing the Selections for Service Type in Event Searches".
  - Service status. See "Customizing the Selections for Service Status in Event Searches".
  - Device types. See "Customizing the Selections for Device Type in Event Searches".

Customizing the Case Sensitivity of Event Searches

The **Match case** check box in the Event Search dialog box is selected, indicating that, by default, event searches are case-sensitive in Customer Center.

To change the default case sensitivity of event searches:

1. Open the *CCSDK_home*/**CustCntr/custom/Customized.properties** file in a text editor, where *CCSDK_home* is the directory in which the Customer Center software development kit (CCSDK) is installed.
2. Add the following entry:

```
par.eventsearch.default.matchcase=false
```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

When you deploy this customization, event searches are not case-sensitive, by default. The **Match case** check box in the Event Search dialog box is not selected.

Customizing the Selections for Service Type in Event Searches

The **Service Type** field in the Event Search dialog box displays the possible selections for service types that can be used in event searches.

To customize the selection for service types in event searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **par.eventsearch.servicetypes.format** entry. For example:

   ```
   par.eventsearch.servicetypes.format={0,choice,1#/service/ip|2#/service/email|3#/
   service/fax|1,default,2}
   ```

   In this example,

   - The service types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default service type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **2**; that is, **/service/email**.

   > **Note:**
   >
   > The default setting for the **Service Type** field is defined by the following statement. (The default service type is **/Not Specified**.)
   >
   > ```
   > par.eventsearch.servicetypes.format={0,choice, 0#(Not Specified)|1#/
   > service|2#/service/ip|3#/service/email|4#/service/telco|5#/service/
   > telco/gsm|6#/service/telco/gsm/sms|7#/service/telco/gsm/telephony|
   > 1,default,0}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Service Status in Event Searches

The **Service Status** field in the Event Search dialog box displays the possible selections for service status that can be used in event searches.

To customize the selection for service status in event searches:

1. Open the CCSDK_home/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **par.eventsearch.servicestatus.format** entry. For example:

   ```
   par.eventsearch.servicestatus.format={0,choice,0#All|10100#Active|10102#Inactive|
   1,default,0}
   ```

   In this example,

   - The service status entries you wish to include are numbered **0# 10100#** and so on, delimited by |.
   - The default service status that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **0**; that is, **All**.

> **Note:**
>
> The default setting for the **Service Status** field is defined by the following statement. (The default service status is **All**.)
>
> ```
> par.eventsearch.servicestatus.format={0,choice,0#All|10100#Active|
> 10102#Inactive|10103#Closed|1,default,0}
> ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Device Type in Event Searches

The **Device Type** field in the Event Search dialog box displays the possible selections for device types that can be used in event searches.

To customize the selection for device types in event searches:

1. Open the *CCSDK_home*/**CustCntr/custom/CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **eventsearch.devicetypes.format** entry. For example:

   ```
   par.eventsearch.devicetypes.format={0,choice,1#/device/num|2#/device/sim|1,default,1}
   ```

   In this example,

   - The device types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default device type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **1**; that is, **/device/num**.

   > **Note:**
   >
   > The default setting for the **Device Type** field is defined by the following statement. (The default device type is **Not Specified**.)
   >
   > ```
   > par.eventsearch.devicetypes.format={0,choice,0#(Not Specified)|1#/
   > device|2#/device/num|3#/device/sim|1,default,0}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing Balance Group Searches

Customer Center has the following criteria to balance group searches for wholesale customers. Customer Center allows you to:

- Set the threshold for the number of available balance groups to display in Customer Center. For more information, see Customer Center Help.
- Customize the case sensitivity of searches for balance groups. See "Customizing the Case Sensitivity of Balance Group Searches".

- Customize searches for balance groups by adding custom settings for the following search components:
  - Service types. See "Customizing the Selections for Service Type in Balance Group Searches".
  - Service status. See "Customizing the Selections for Service Status in Balance Group Searches".
  - Device types. See "Customizing the Selections for Device Type in Balance Group Searches".

Customizing the Case Sensitivity of Balance Group Searches

The **Match case** check box in the Balance Group Search dialog box is selected, indicating that, by default, balance group searches are case-sensitive in Customer Center.

To change the default case sensitivity of balance group searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**Customized.properties** file in a text editor, where *CCSDK_home* is the directory in which the Customer Center software development kit (CCSDK) is installed.
2. Add the following entry:

   ```
   balancegroupsearch.default.matchcase=false
   ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

When you deploy this customization, balance group searches are not case-sensitive, by default. The **Match case** check box in the Balance Group Search dialog box is not selected.

Customizing the Selections for Service Type in Balance Group Searches

The **Service Type** field in the Balance Group Search dialog box displays the possible selections for service types that can be used in balance group searches.

To customize the selection for service types in balance group searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **balancegroupsearch.servicetypes.format** entry. For example:

   ```
   balancegroupsearch.servicetypes.format={0,choice,1#/service/ip|2#/service/email|3#/service/fax|1,default,2}
   ```

   In this example,
   - The service types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default service type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **2**; that is, **/service/email**.

> **Note:**
>
> The default setting for the **Service Type** field is defined by the following statement. (The default service type is **/service**.)
>
> ```
> balancegroupsearch.servicetypes.format={0,choice, 0#(Not Specified)|1#/
> service|2#/service/ip|3#/service/email|4#/service/telco|5#/service/
> telco/gsm|6#/service/telco/gsm/sms|7#/service/telco/gsm/telephony|
> 1,default,1}
> ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Service Status in Balance Group Searches

The **Service Status** field in the Balance Group Search dialog box displays the possible selections for service status that can be used in balance group searches.

To customize the selection for service status in balance group searches:

1. Open the *CCSDK_home*/**CustCntr/custom/CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **balancegroupsearch.servicestatus.format** entry. For example:

   ```
   balancegroupsearch.servicestatus.format={0,choice,0#All|10100#Active|10102#Inactive|
   1,default,0}
   ```

   In this example,

   - The service status entries you wish to include are numbered **0# 10100#** and so on, delimited by |.
   - The default service status that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **0**; that is, **All**.

   > **Note:**
   >
   > The default setting for the **Service Status** field is defined by the following statement. (The default service status is **Active**.)
   >
   > ```
   > balancegroupsearch.servicestatus.format={0,choice,0#All|10100#Active|
   > 10102#Inactive|10103#Closed|1,default,10100}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Device Type in Balance Group Searches

The **Device Type** field in the Balance Group Search dialog box displays the possible selections for device types that can be used in balance group searches.

To customize the selection for device types in balance group searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **balancegroupsearch.devicetypes.format** entry. For example:

   ```
   balancegroupsearch.devicetypes.format={0,choice,1#/device/num|2#/device/sim|
   1,default,1}
   ```

   In this example,

   - The device types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default device type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **1**; that is, **/device/num**.

   > **Note:**
   >
   > The default setting for the **Device Type** field is defined by the following statement. (The default device type is **Not Specified**.)
   >
   > ```
   > balancegroupsearch.devicetypes.format={0,choice,0#(Not Specified)|1#/
   > device|2#/device/num|3#/device/sim|1,default,0}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing Product/Discount Searches

Customer Center has the following criteria to product/discount searches for wholesale customers. Customer Center allows you to:

- Set the number of available products/discounts to display in Customer Center. For more information, see Customer Center Help.
- Customize the case sensitivity of searches for products/discounts. See "Customizing the Case Sensitivity of Product/Discount Searches".
- Customize searches for products/discounts by adding custom settings for the following search components:
  - Service types. See "Customizing the Selections for Service Type in Product/Discount Searches".
  - Service status. See "Customizing the Selections for Service Status in Product/Discount Searches".
  - Device types. See "Customizing the Selections for Device Type in Product/Discount Searches".

Customizing the Case Sensitivity of Product/Discount Searches

The **Match case** check box in the Product/Discount Search dialog box is selected, indicating that, by default, product/discount searches are case-sensitive in Customer Center.

To change the default case sensitivity of product/discount searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**Customized.properties** file in a text editor, where *CCSDK_home* is the directory in which the Customer Center software development kit (CCSDK) is installed.
2. Add the following entry:

   ```
   proddiscsearch.default.matchcase=false
   ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

When you deploy this customization, product/discount searches are not case-sensitive, by default. The **Match case** check box in the Product/Discount Search dialog box is not selected.

Customizing the Selections for Service Type in Product/Discount Searches

The **Service Type** field in the Product/Discount Search dialog box displays the possible selections for service types that can be used in product/discount searches.

To customize the selection for service types in product/discount searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **proddiscsearch.servicetypes.format** entry. For example:

   ```
   proddiscsearch.servicetypes.format={0,choice,1#/service/ip|2#/service/email|3#/
   service/fax|1,default,2}
   ```

   In this example,

   - The service types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default service type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **2**; that is, **/service/email**.

   > **Note:**
   >
   > The default setting for the **Service Type** field is defined by the following statement. (The default service type is **/service**.)
   >
   > ```
   > proddiscsearch.servicetypes.format={0,choice,1#/account|2#/service|3#/
   > service/ip|4#/service/email|5#/service/telco|6#/service/telco/gsm|7#/
   > service/telco/gsm/sms|8#/service/telco/gsm/telephony|1,default,2}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Service Status in Product/Discount Searches

The **Service Status** field in the Product/Discount Search dialog box displays the possible selections for service status that can be used in product/discount searches.

To customize the selection for service status in product/discount searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **proddiscsearch.servicestatus.format** entry. For example:

```
proddiscsearch.servicestatus.format={0,choice,0#All|10100#Active|10102#Inactive|
1,default,0}
```

In this example,

- The service status entries you wish to include are numbered **0# 10100#** and so on, delimited by |.
- The default service status that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **0**; that is, **All**.

> **Note:**
>
> The default setting for the **Service Status** field is defined by the following statement. (The default service status is **Active**.)
>
> ```
> proddiscsearch.servicestatus.format={0,choice,0#All|10100#Active|
> 10102#Inactive|10103#Closed|1,default,10100}
> ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Device Type in Product/Discount Searches

The **Device Type** field in the Product/Discount Search dialog box displays the possible selections for device types that can be used in product/discount searches.

To customize the selection for device types in product/discount searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **proddiscsearch.devicetypes.format** entry. For example:

   ```
   proddiscsearch.devicetypes.format={0,choice,1#/device/num|2#/device/sim|1,default,1}
   ```

   In this example,

   - The device types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default device type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **1**; that is, **/device/num**.

   > **Note:**
   >
   > The default setting for the **Device Type** field is defined by the following statement. (The default device type is **Not Specified**.)
   >
   > ```
   > proddiscsearch.devicetypes.format={0,choice,0#(Not Specified)|1#/device|
   > 2#/device/num|3#/device/sim|1,default,0}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing Service Searches

Customer Center has the following criteria for service searches. Customer Center allows you to:

- Set the threshold for the number of available services to display in Customer Center. For more information, see Customer Center Help.
- Customize the case sensitivity of searches for services. See "Customizing the Case Sensitivity of Service Searches".
- Set the step size for the searches. See "Customizing the Step Search Size".
- Customize service searches by adding custom settings for the following search components:
  - Service types. See "Customizing the Selections for Service Type in Service Searches".
  - Service status. See "Customizing the Selections for Service Status in Service Searches".
  - Device types. See "Customizing the Selections for Device Type in Service Searches".

Customizing the Case Sensitivity of Service Searches

The **Match case** check box in the Service Search dialog box is selected, indicating that, by default, service searches are case-sensitive in Customer Center.

To change the default case sensitivity of service searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**Customized.properties** file in a text editor, where *CCSDK_home* is the directory in which the Customer Center software development kit (CCSDK) is installed.
2. Add the following entry:

   ```
   servicesearch.default.matchcase=false
   ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

When you deploy this customization, service searches are not case-sensitive, by default. The **Match case** check box in the Service Search dialog box is not selected.

Customizing the Step Search Size

You can customize the step search size for retrieving services.

To customize the step search size:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**Customized.properties** file in a text editor, where *CCSDK_home* is the directory in which the Customer Center software development kit (CCSDK) is installed.
2. Set the **servicesearch.stepsize** entry to the appropriate value for your server (memory) configuration. The default is **100**.
3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Service Type in Service Searches

The **Service Type** field in the Service Search dialog box displays the possible selections for service types that can be used in service searches.

To customize the selection for service types in service searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **servicesearch.servicetypes.format** entry. For example:

   ```
   servicesearch.servicetypes.format={0,choice,1#/service/ip|2#/service/email|3#/
   service/fax|1,default,2}
   ```

   In this example,

   - The service types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default service type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **2**; that is, **/service/email**.

   > **Note:**
   >
   > The default setting for the **Service Type** field is defined by the following statement. (The default service type is **/service**.)
   >
   > ```
   > proddiscsearch.servicetypes.format={0,choice,1#/service|2#/service/ip|
   > 3#/service/email|4#/service/telco|5#/service/telco/gsm|6#/service/
   > telco/gsm/sms|7#/service/telco/gsm/telephony|1,default,1}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Service Status in Service Searches

The **Service Status** field in the Service Search dialog box displays the possible selections for service status that can be used in service searches.

To customize the selection for service status in service searches:

1. Open the *CCSDK_home*/**CustCntr**/**custom**/**CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **servicesearch.servicestatus.format** entry. For example:

   ```
   servicesearch.servicestatus.format={0,choice,0#All|10100#Active|10102#Inactive|
   1,default,0}
   ```

   In this example,

   - The service status entries you wish to include are numbered **0# 10100#** and so on, delimited by |.
   - The default service status that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **0**; that is, **All**.

> **Note:**
>
> The default setting for the **Service Status** field is defined by the following statement. (The default service status is **Active**.)
>
> ```
> servicesearch.servicestatus.format={0,choice,0#All|10100#Active|
> 10102#Inactive|10103#Closed|1,default,10100}
> ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Customizing the Selections for Device Type in Service Searches

The **Device Type** field in the Service Search dialog box displays the possible selections for device types that can be used in service searches.

To customize the selection for device types in service searches:

1. Open the *CCSDK_home*/**CustCntr/custom/CustomizedResources.properties** file in a text editor.
2. Add the custom setting for the **servicesearch.devicetypes.format** entry. For example:

   ```
   servicesearch.devicetypes.format={0,choice,1#/device/num|2#/device/sim|1,default,1}
   ```

   In this example,

   - The device types you wish to include are numbered **1# 2#** and so on, delimited by |.
   - The default device type that is displayed when you access the dialog box is specified as **1,default,***n*. In this example, the default is **1**; that is, **/device/num**.

   > **Note:**
   >
   > The default setting for the **Device Type** field is defined by the following statement. (The default device type is **Not Specified**.)
   >
   > ```
   > servicesearch.devicetypes.format={0,choice,0#(Not Specified)|1#/device|
   > 2#/device/num|3#/device/sim|1,default,0}
   > ```

3. Save the file.
4. Build the custom **.jar** file and deploy this customization.

Hiding the Password Fields in Customer Center

Hiding the password fields for services in Customer Center (in the **Customize Services** page and the **Services** tab) enhances security. When the password fields are hidden, the BRM server generates the password for services. You cannot hide the password fields for the **/service/admin_client** and **/service/pcm_client** service types.

To hide the password fields for services in Customer Center:

1. Open the *CCSDK_home***/CustCntr/custom/Customized.properties** file in a text editor.
2. Add the following line after the comment statements:

```
nonCSRservices.hide.passwordfields=value
```

where *value* is:

- **true** to hide the password fields for service types other than **/service/admin_client** and **/service/pcm_client**.
- **false** to display the password fields. This is the default.

3. Save and close the file.
4. Build the custom **.jar** file, and deploy this customization.
5. Restart Customer Center.

Disabling the Child Amounts Check Box

To disable the include **Child Amounts** check box in Customer Center:

1. Open the **Customized.properties** file in the *CCSDK_Home*/**CustCntr/custom** directory.
2. After the comment statements, add the following line:

```
Customized.ReadOnlyIncludeChild=true
```

3. Save and close the file.

Adding a Custom Service as a Login to Customer Center

As an alternative to **/service/admin_client**, you can add a custom service as a login.

To add a custom service as a login to Customer Center:

1. Open the *CCSDK_Home*/**CustCntr/custom/Customized.properties** file in a text editor.

   Alternatively, you can add the line to **WizardCustomizations.properties**.
2. Add the following line after the comment statements:

```
customercenter.loginType=/service/CustomService
```

3. Save and close the file.
4. Build the custom **.jar** file and deploy this customization.
5. Restart Customer Center.

Using Customer Center SDK Scripts for Customer Center

Table 47-1 contains the scripts provided with Customer Center SDK to automate many of the processes for preparing and building your customizations:

Table 47-1 Customer Center SDK Scripts

Script	Purpose
runConfigurator	Runs the Configurator client. See "Using Configurator".

Table 47-1 (Cont.) Customer Center SDK Scripts

Script	Purpose
runFldSpecWidget	Runs a read-only Storable Class Editor widget for reading BRM field definitions to copy and use for **modelFieldDescription**, **displayFieldDescription**, and **displayFieldFormat** entries for new fields. You can use these field definitions when you write code to add functionality to the Customer Center and Self-Care Manager applications.
makecertificate	Creates a self-signed certificate for deploying your customizations. See "Creating a Self-Signed Java Security Certificate".
runCustomerCenter	Runs a standalone version of Customer Center for local testing of your customizations. This script requires a connection to a CM, but doesn't require a Web server. See "Testing Your Customizations".
compile	Recompiles the example source files in all subdirectories under the **/CustCntrExamples** directory. Use this script before you run the example source files. **Tip**: To see how the **buildAll** script compiles, packages, and signs custom source code, copy **Customized.properties** and **CustomizedResources.properties** files from the example's folder to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory.
testExample	Tests the compiled Customer Center examples in the following directories under the **/CustCntrExamples** directory: • **AccountCreation** • **General** • **Notes** • **Service** • **Summary** • **Profile** • **Events**
unpackHelp	Unpacks the Customer Center Help file, **Customer_Center_Help_en.jar**.
buildHelp	Builds the **ccCustomHelp_en.jar** file for distributing your customized online help.

Adding New Pages to the Customer Center Interface

This section describes the concepts and components you need to add pages for account maintenance or for the New account wizard.

To add new pages to Customer Center, you need:

- Experience with a Java graphical user interface (GUI) builder, such as JBuilder
- Experience in building Java applications
- Familiarity with "Understanding the BRM Business Application SDK Framework" and "Advanced Customer Center Concepts"

This section includes these topics:

- About Portal Infranet Aware Widgets
- Adding Account Maintenance Pages
- Adding New Account Wizard Pages

About Portal Infranet Aware Widgets

Custom Center uses the Portal Infranet Aware (PIA) widget set when building a custom page. These widgets include extra APIs that know about BRM; they are capable of automatically toggling currency data, some are capable of exporting their data to HTML, and others contain bug fixes that you need when using the normal Swing widgets. These widgets are in the **com.portal.bas.comp** Java package in **basacAll.jar**. The most important mappings from PIA widgets to Swing widgets include:

- PIACustomizablePanel == JPanel
- PIAScrollPane == JScrollPane
- PIATextField == JTextField
- PIAReadOnlyField == JLabel
- PIASpecSpreadSheet == JTable

When constructing a custom page, always try to use the PIA widget set for the reasons described above.

For more information on PIA widgets, see:

- Components Used in Customer Center
- Advanced Customer Center Concepts
- Customer Care API reference

Adding Account Maintenance Pages

This section provides information and implementation tips on adding account maintenance pages.

- Overview of Account Maintenance Components
- Saving Changes
- Refreshing Data in the UI
- Currency Toggling
- Drill-Down Links
- Advanced Drill-Down Techniques
- Modifying the Customer Center Permissions
- Adding Your Page to the Customer Center Toolbar

Overview of Account Maintenance Components

When working with an existing account, the Customer Center framework uses the **PCustomerCenterContext** class. This class contains APIs for accessing the current account and for accessing global data managers. Each open account contains a further set of APIs encapsulated in the **PAccountViewContext** class. This class contains APIs for implementing drill-downs and accessing the refresh manager for the account.

Every page in the account maintenance view, including tabs and drill-down panels, must implement the **PAccountViewPage** interface. This interface contains methods required by the Customer Center framework to interact with the various pages in the tabbed pane. If you are creating a new page from scratch, it is highly recommended that you start with

PMaintenancePage. This base class implements the **PAccountViewPage** interface and provides a basic implementation in many of the methods.

If you choose to override a particular method in **PMaintenancePage**, simply call the superclass method first to take advantage of the default implementation.

If you start with **PMaintenancePage**, you should be aware of these methods:

- **save()**: Saves data on a page. Provide code in this method to save the data in your page.

  This method is called automatically by the framework.
- **hasUnsavedChanges()**: This method is called automatically by the framework to determine if your page needs to be saved.
- **getLabel()** – If your page contains links allowing the user to drill down within your custom tab, you provide a string in this method that describes your page. This string appear as the initial tag in the breadcrumb trail.
- **refresh()**: This method is called when the user clicks the **Refresh** toolbar button. You should refresh the data on your page in this method, going to BRM as required.
- **recycle()**: Pages in the tabbed pane are reused between accounts. In this method, clear out all data you set manually so data from one account doesn't accidentally appear when displaying the next account.

If you are using the BAS PIA widgets on your screen, you might want to use the **collectData()** method when gathering data for saving. For example:

```
PModelHandle modl = PClientContext.getServices().
        createNewModel(PModelHandle.UNTYPED);
PCollectDataEvent evt = new PCollectDataEvent(
        this, PCollectDataEvent.FOR_DIRTY, modl);
collectData(evt);
```

After this code is invoked, the **modl** variable represents data that was modified on your page. You can now convert this data from a **PModelHandle** into an flist and pass it off to your appropriate opcode. The conversion is performed as shown in this example:

```
PCachedContext conn = (PCachedContext)
        CCCompatibilityUtility.getConnection();

//This is how you turn a PModelHandle into an FList
        FList flist = (FList)conn.lookupModel(modl);
```

Saving Changes

When a user leaves a Customer Center page, the page saves its own data.

> **Note:**
>
> Customer Center users leave a page by changing tabs, drilling down to another page, or using the **Back** button.

The act of leaving a page invokes code that gathers the data and calls the opcode used for saving the data. Each page formats the appropriate input flist and calls the correct opcode. If a data entry error occurs, a page can use the global **PSaveManager** methods to notify the user of the error and locate the field causing the error. The global **SaveManager** contains

convenience methods that the pages can use for alerting the user and locating the UI field causing the error.

To save changed data from a page, the framework calls the **hasUnsavedChanges()** method on the **PAccountViewPage** interface. If the class returns **true**, the **save()** method is called on the page. If the page signals a save failure, a **PSaveException** is returned.

Customer Center uses wrapper opcodes to group editable data together so that all changes can be saved with a single opcode call. For example, the CUST_UPDATE_CUSTOMER opcode is a wrapper opcode that calls CUST_MODIFY_PROFILE. This feature allows you to embed a panel for a **/profile** object into a panel that displays regular account data, such as contact or payment data, and save any changes using a single opcode call. Profile panels do not have to be contained within separate tabs.

> **Note:**
>
> Avoid calling multiple opcodes when saving page data. If you call more than one opcode, you might need to roll back a set of changes if a subsequent page save attempt fails.

If your page is constructed using the BAS Portal Infranet-aware (PIA) widgets, you probably do not need to override this method.

The base implementation in **PMaintenancePage** returns the appropriate value. However, if you are using standard Swing widgets in your page, you probably need to determine if your page must be saved or not:

- If your page requires saving, return **true**.
- If your page doesn't require saving, return **false**. The method is not be called.

If you are subclassing (directly or indirectly) a **PIACustomizablePanel** in your custom page, call the **setInputTracking(true)** method after constructing the UI for your page. This is especially important if you are using **collectData()** to gather the data for saving. Input tracking tells the system that every PIA widget should track changes made by a user. (This feature displays the colored widgets when a user makes a change). When **collectData()** is called, it asks each widget if it has been modified. If it has been modified, it retrieves the modified data.

> **Note:**
>
> If input tracking is turned off, the widgets won't know that they have been changed.

Refreshing Data in the UI

When components use one of the **PRefreshManager** listener methods to register for event notification, they are notified whenever a data type of interest changes.

PRefreshManager tracks the following types of data changes:

- Balance impacts
- Contact information
- Changes to credit limits

- Deferred actions
- Hierarchy changes
- Holdings changes (product and deal purchases and cancellations)
- Payment type changes
- Service changes (login, status, password, and deferred action count)
- Status changes

There is a separate refresh manager instance created for each open account in Customer Center so that changes in one account do not accidentally impact an unrelated account. The **PRefreshManager** instance associated with an open account can be retrieved from the account's view context by using the **getRefreshManager()** method in **PAccountViewContext**.

Each **PAccountViewPage** instance also contains a **refresh()** method that is called when the user clicks the **Refresh** toolbar button. The **refresh()** method is invoked for a page if the page is currently active. Each page should update its data when this method is invoked.

Use the **PRefreshManager** component's refresh mechanism if you are:

- Adding a page that displays data found in other pages.
- Adding a page that displays data that is directly impacted by changes elsewhere in the interface.

  If you change a data type, call one of the **PRefreshManager** process methods so that other registered components are also updated as required. See for **PRefreshManager** for details about the various listener and process methods supported by the default implementation.

You can implement the **refresh()** method by resetting the **PModelHandle** on the page as follows:

```
setModelHandle(getModelHandle());
```

Currency Toggling

If you use multiple currencies in an account, you can implement currency toggling. When you implement currency toggling on your panel, use the Portal Infranet-aware (PIA) widget set when building a new page. The PIA widget set is part of the **com.portal.bas.comp** package.

The PIA widgets are wrappers for common **Swing** widgets. When setting the data for any PIA widget, use the **setLightData** API instead of the standard **Swing** methods such as **setText**. For example, if you have a screen that displays balance information, such as **Balance: $15.99**, you could make the **Balance** widget a **JLabel** and the **$15.99** widget a **PIAReadOnlyField**.

> **Note:**
>
> If you are specifying **displayFieldDescription** for your widgets, this happens automatically.

Call **setCurrencyDisplay(true)** on each widget displaying currency data.

> **Note:**
>
> This API is available for the PIA widgets that are capable of displaying currency data. For more information, see *Customer Care API Reference*.

Drill-Down Links

The Customer Center interface allows users to *drill down* from one page to another. The **PCCLink** class provides the drill down functionality, including the visual element to display your link. You create a new **PCCLink** as follows:

```
new PCCLink("text", pageClass, "trailTag", null|"parent");
```

Table 47-2 describes the input parameters for a new **PCCLink**:

Table 47-2 Input Parameters for PCCLink

Input Parameter	Description
text	The text you want to appear in the link widget itself. Include the quotes.
pageClass	A quoted string that contains the name of the class you want the link to drill down to.
trailTag	The text label, in quotes, that you want to appear in the bread crumb trail above the tabbed panel.
parent	The name of the destination tab. This tag is the same tag specified in **CustomerCenter.properties**. Leave the value as **null** if you do not want to drill down to a different page. Include quotes if the value is not **null**. Use this parameter to drill down from one page tab to another.

Drill-down example 1

To create a **PCCLink** that drills down from the Summary page to the status change screen, include the following link on the Summary page:

```
new PCCLink("Status", com.portal.app.cc.PChangeStatusPage, "Change Status", null);
```

The **PCCLink** automatically calls the framework API required for drilling down a link selected by the user. You do not need to call any APIs or hook up event handlers.

Drill-down example 2

To make a drill down that appears as if a user had drilled down directly from the **Service** tab, include the following link on the Summary page:

```
new PCCLink("Status", com.portal.app.cc.PChangeStatusPage, "Change Status","service");
```

When this link is selected, the **Service** tab moves to the front and the bread crumb trail on the displayed page reads **Service -> Change Status**.

See the **DrillDownTest.java** sample code in *CCSDK_home*/**CustomerCareSDK/CustCntrExamples** for a detailed example.

Invoking the drill-down API directly

You can gain further control over drill-down behavior by invoking the drill-down API directly rather that using the **PCCLink** class.

To invoke drill downs directly:

1. Retrieve the context for the current account view. The context is a reference to the tabbed pane for the currently active account.

   You can access the **PAccountViewContext** through the **PCustomerCenterContext**:

   ```
   CCCompatibilityUtility.getCustomerCenterContext().getAccountViewContext();
   ```

2. From the **PAccountViewContext**, call one of the four **switchToPage** API variants.

   You can either pass in a pre-existing **PAccountViewPage** instance or a reference to the **Class** instance of the page. If a reference is passed, the framework first attempts to find an instance of this class within the current view. If one is found, it is reused and a handle to the existing **PAccountViewPage** is returned. If no instance is found, a new one is created and a handle to the new **PAccountViewPage** instance is returned.

> **Tip:**
>
> If possible, use the **switchToPage** method that takes a **Class** instance. If you pass a **Class** instance, the framework tracks the page instance for you. If you do need to call any specific API on the page you are drilling down to, an instance is returned to you from the **switchToPage** method to enable the drill-down. For more information, see *Customer Care API Reference* for information on this version of the **switchToPage** method.

Drill-down simulation

You can simulate two types of drill downs:

- You can drill down within the current tab.
- You can switch to a different tab and simulate a drill down from the top-level page in that tab.

Two of the **switchToPage()** methods take three parameters that allow you to drill down within the current tab or switch to a different tab and simulate a drill down from the new top-level tab. Pass in the tag reference for the tab you wish to drill down to, and the framework automatically switches the active tab before performing the drill down.

> **Note:**
>
> The tag reference for each tab is the same tag specified in the **custinfo.tabs** property in **CustomerCenter.properties**.

For more information on using the **switchToPage()**, see "Advanced Drill-Down Techniques".

Drill-down implementation procedure

To implement a drill down:

1. Pass in the following:
   - The label to display in the breadcrumb trail for the new top-level tab.

- The **Class** name of the parent page to display as a drill-down. If an instance of that class exists, it becomes the current drill-down. Otherwise, the class is instantiated and added as a drill-down.
- A **String** with the name of the parent component to drill down from.

The framework automatically switches the active tab before performing a drill-down. The label text used for each tab is the same as specified in the **custinfo.tabs** property in **CustomerCenter.properties**.

2. Provide a graphical element to invoke the **switchToPage** method. Normally you would use the **Link** widget and hook up an **ActionListener** to it. Within the **ActionListener** callback method, **actionPerformed**, retrieve the context for the current account and invoke the **switchToPage** method.

To embed a link within a table cell, use the framework API but create a renderer for your table column instead of using the **PCCLink** or **Link** widgets directly. Instead, use **PLinkRenderer**. For an example, see the **DrillDownTest.java** sample code in the **General** directory under *CCSDK_home*/**CustomerCareSDK/CustCntrExamples**.

Advanced Drill-Down Techniques

To gain more control over drill downs, you can invoke the drill down API yourself instead of using **PCCLink()**.

1. Retrieve the context for the current account view.

   The context is a reference to the tabbed pane for the currently active account. You can access the **PAccountViewContext** through the **CustomerCenterContext**:

   ```
   CCCompatibilityUtility.getCustomerCenterContext().getAccountViewContext();
   ```

2. From the **PAccountViewContext**, call one of the four variants of the **switchToPage()** API. You can either pass in a pre-existing **PAccountViewPage** instance or a reference to the class instance of the page.

If you pass in a class instance, the framework first attempts to find an instance of this class within the current view. If one is found, the framework re-uses it and return a handle to it. If no instance is found, a new one is created and a handle is returned.

> **Tip:**
>
> If you use the **switchToPage()** method that takes a class instance, the framework keeps track of the page instance for you. Also, if you do need to call any specific API on the page you are drilling down to, an instance is returned to you from the **switchToPage()** method to accomplish this.

You must provide a graphical element to invoke the **switchToPage()** method. You typically use the **com.portal.ctrl.Link** widget, and hook up an **ActionListener** to it. Within the **ActionListener** callback method (**actionPerformed**), retrieve the context for the current account and invoke the **switchToPage()** method.

To embed a link within a table cell, you use the framework API above. However, instead of using a PCCLink or Link widget directly, you create a renderer for your table column. You can use the com.portal.app.comp.PLinkRenderer for this purpose. There is a sample of how to use this renderer in the SDK examples directory.

Modifying the Customer Center Permissions

Customer Center SDK includes the **PRestriction** class that defines server-side permissions for the user or the client application to perform certain functions.

Adding Your Page to the Customer Center Toolbar

You can add your page to the account maintenance toolbar button instead of as a separate tab. This section includes information on how to present your data in a floating dialog box, such as for notes, that the user can keep up at all times.

Write your toolbar button code as you do when writing tab code. You can extend **PMaintenancePage**, but that is not required, and you can start with any base class. However, if you plan on using BAS PIA widgets you must start with a **PIACustomizablePanel**. Once you have your page, you create an action class for the toolbar. Your class should extend the BAS action class as follows:

```
public class WorkFlowAction extends PBASAction {
    public WorkFlowAction(PClientComponent comp) {
        super(comp, "workflow");
}
```

Note the **workflow** tag highlighted above. This is a mapping into the property file indicating attributes of your toolbar button such as label, icon, and so forth. Override the **actionPerformed()** method and display your dialog when this method is invoked:

```
public void actionPerformed(ActionEvent ae) {
}
```

To add your custom toolbar button to Customer Center, modify **Customized.properties** and **CustomizedResources.properties**. For example, if you identify your additional button by the **workflow** tag, your property file changes is as follows:

Customized.properties

```
customercenter.toolbar=back home SEP VSEP save refresh SEP VSEP SEP newaccounts SEP VSEP
  search notes workflow
customercenter.tb.class.workflow=WorkFlowAction
```

CustomizedResources.properties

```
customercenter.workflow.label=
customercenter.workflow.icon=myWorkFlow.gif
customercenter.workflow.desc=Launches the workflow dialog
customercenter.workflow.accel=
customercenter.workflow.mnemonic=
```

> **Tip:**
>
> You can optionally fill in values for parameters that have none. For example, if you fill in the **label** property, a label appears next to your toolbar button.

Adding New Account Wizard Pages

This section includes concepts and guidelines for adding pages to the new accounts wizard:

- Understanding the New Accounts Wizard
- Base Storable Classes for Account Creation Pages
- Methods Used in New Account Creation Pages

Understanding the New Accounts Wizard

This section describes how the new account wizard works.

PWizardPage

Every page in the new account wizard must implement the **PWizardPage** interface. This interface contains methods required by the wizard framework. For more information on the **PWizardPage** interface, see the *Customer Care API Reference*.

> **Tip:**
>
> If you add a new account wizard page to using Configurator, all entries are written out for you automatically.

Data sharing

The new account wizard in Customer Center uses *shared data*. Shared data allows one page to access data from another page, or for one page to broadcast data changes to other pages by using data registration.

Pages in the wizard can use the shared data in a request-based or broadcast-based manner:

- **Request-based shared data.** When a user switches to a page, the page can retrieve the shared data.
- **broadcast-based data sharing.** If a page must be notified when a piece of data (such as the primary and secondary currency) changes, it can register for notification with the shared data manager.

Each wizard has a context that can be retrieved from the main **CustomerCenterContext**. Only one wizard can be active at any given time so the **getWizardContext()** method can be used to access the active wizard. If it is null, a wizard is not currently active. From the wizard context, you can access the **SharedDataManager**. Use this class if you are interested in registering for notification of data changes, or simply to access the shared data object itself.

You do not need to write a page if you want to tap into the shared data mechanism. See the **AcctCreationTapNonGUI.jav**a example in the *CCSDK_home***/CustomerCareSDK/CustCntrExamples/AccountCreation** directory. This example demonstrates how to retrieve shared data when starting the wizard.

Partial validation

The New Account wizard performs partial validation on all pages contained within it. Data is sent to BRM on a per-page basis as the user moves between pages. This mechanism catches CSR data entry errors. Each page within the wizard validates its own data and identifies any errors encountered while validating and saving the data in BRM.

When the user attempts to leave a page, the page's **validateWizardPage()** method is called. This signals the page to gather its data and send it to BRM for validation. A number of convenience methods that assist in performing validation are available in the **WizardValidationManager**.

> **Note:**
>
> The **WizardValidationManage**r is available through the wizard context.

For example, if a page doesn't have any special validation needs of its own, it can immediately call the **validatePage API** on the **WizardValidationManager**. This method performs default data collection on the wizard page, invokes the CUST_VALIDATE_CUSTOMER opcode in BRM, and performs default error handling when if validation fails. Alternatively, the page can collect its own data and, perhaps after examining it, call the convenience method **validateModel** on the **WizardValidationManager**. This method invokes the CUST_VALIDATE_CUSTOMER opcode and, if validation failed, hands an error data structure back to the page. The page can then pass the error information to an API in the **SaveManager** that attempts to identify the field(s) that caused the error.

Pages can be coded to perform data validation and any error determination itself without using the **WizardValidationManager**, the **SaveManager**, or even without calling an **opcode**.

If an error results when the user clicks **Finish** to create a new account based upon the entered data, the wizard can call the **handleWizardCommitError** method on each wizard page successively until a page reports back that it wants to handle the error. That page is then made visible. To handle the error, the page could then:

- Call a convenience method on the **WizardValidationManager** to get the default commit-time error handling behavior.
- Use the **SaveManager** method mentioned above to try and identify any fields that were responsible for the error.
- Perform its own analysis of the error data.

Base Storable Classes for Account Creation Pages

You can start with any base class you wish when creating your account creation page.

> **Tip:**
>
> Customer Center SDK does not include a base class for adding account creation pages. However, you can optionally use **PMaintenancePage** class described in "Adding Account Maintenance Pages", especially if you want to use the same page as a tab during account maintenance.

Methods Used in New Account Creation Pages

These methods are used for account creation panels:

- **getWizardHelpID()**: Use this method to add custom help to your page. Include the empty string ("") if you are not including customized help.
- **recycle()**: Pages in the new accounts wizard are reused between accounts. In this method, clear out all data you set manually so data from one account doesn't accidentally appear when displaying the next account.
- **validateWizardPage()**: This method is called as the user leaves your page in the New Account wizard. It includes a hook to validate any data the user has entered. Since your

page is custom, determine if BRM can validate your page or if you must validate it manually.

If you have modified BRM policy code to validate your page, the following code may be useful:

```
CustomerCenterContext ctx = CCCompatibilityUtility.getCustomerCenterContext();
if (ctx.getWizardContext() instanceofPAccountCreationWizardContext) {
    PAccountCreationWizardContext wizContext =
        PAccountCreationWizardContext) ctx.getWizardContext();
    WizardValidationManager wizValidation =
        wizContext.getWizardValidationManager();
    wizValidation.validatePage();
}
```

This code utilizes the convenience class **WizardValidationManager** to handle validation. However, your page must use the BAS PIA widget set to take full advantage of this mechanism.

- **handleWizardCommitError()**: This method is called when the user attempts to create the account and an error occurs. A data structure is passed in containing a description of the error. Return **true** if the error actually occurs on your page. If the error is not yours, return **false**. When you return **true**, the wizard framework automatically takes the user to your page.

  > **Tip:**
  >
  > Visually indicate to the user which field is causing the error. If you are using the BAS PIA widgets, the following code might be useful:

```
public boolean handleWizardCommitError(CustomerError error) {
    CustomerCenterContext ctx = CCCompatibilityUtility.getCustomerCenterContext();
    if (ctx.getWizardContext() instanceofPAccountCreationWizardContext) {
        PAccountCreationWizardContext wizContext = (PAccountCreationWizardContext)
          ctx.getWizardContext();
        WizardValidationManager wizValidation = wizContext.getWizardValidationManager();
        return wizValidation.handleCommitError(error, this);
    } else {
        return false;
    }
}
```

Removing a Payment Method from Customer Center

Removing a payment method prevents it from appearing in the Account Creation wizard, the **Payments** section, the **Hierarchy** section, the **Purchase** wizard, and dialog boxes.

> **Note:**
>
> Testing and maintaining customized code is your responsibility.

To remove a payment method from Customer Center:

1. Open the *CCSDK_Home*/**CustCntr/custom/Customized.properties** file in a text editor.
2. Add the following after the comment statements:

3.
```
custinfo.payment.class=com.portal.custom.PCustomPaymentPage
Customized.hierarchyMoveDlg.class=com.portal.custom.PCustomHierarchyMoveDlg
com.portal.app.ccare.comp.PPaymentPanelBeanImpl.subclass=com.portal.custom.PCustomPay
mentPanelBeanImpl
Customized.BillUnitHierarchyPage.class=com.portal.custom.PCustomBillUnitHierpage.
```

4. Create a **com.portal.custom.PCustomPaymentPage** that contains the following:

```
package com.portal.custom;
.
import com.portal.app.cc.PPaymentPage;
import java.util.Vector;
.
public class PCustomPaymentPage extends PPaymentPage{
    public PCustomPaymentPage(){
        super();
    }
    public Vector getPayMethods () {
        Vector vec = super.getPayMethods();

    //Remove the entry for bad payment method from the vector and return

        return vec;
    }
}
```

5. Create a **com.portal.custom.CustomHierarchyMoveDlg** page that contains the following:

```
package com.portal.custom;
.
import com.portal.app.cc.PHierarchyMoveConfirmDlg
import java.util.Vector;
.
import javax.swing.JFrame;
.
public class PCustomHierarchyMoveDlg extends PHierarchyMoveConfirmDlg {
.
    public PCustomHierarchyMoveDlg(JFrame frame, String title, boolean modal)
{
        super(frame, title, modal);
        // TODO Auto-generated constructor stub
    }
.
    public Vector getPayMethods() {
        Vector vec = super.getPayMethods();
        //Remove the entry for the bad payment method from the vector
                        //and return
        return vec;
    }
}
```

6. Create a **com.portal.custom.PCustomPaymentPanelBeanImpl** page that contains the following:

```
package com.portal.custom;
.
import com.portal.app.cc.BalanceGroupTreeTable;
import com.portal.app.cc.PPayType;
import com.portal.app.ccare.comp.PPaymentPanelBeanImpl;
import java.util.Iterator;
import java.util.Vector;
.
public class PCustomPaymentPanelBeanImpl extends PPaymentPanelBeanImpl {
```

```
.
    public PCustomPaymentPanelBeanImpl() {
        super();
    }
.
    public void refreshPaymentMethodModel(BalanceGroupTreeTable
balanceTreeTable) {
        super.refreshPaymentMethodModel(balanceTreeTable);
        Vector payMethod = getPayMethod();
        //Remove the entry for the bad payment method from the vector
        //and return
.
        setPayMethod(payMethod);
.
    }
.
    public Vector getPayTypes() {
        Vector list = super.getPayTypes();
        int i = 0;
        int index = -1;
        //Remove the entry for the bad payment method from the vector
        //and return
.
        for (Iterator iterator = list.iterator(); iterator.hasNext();) {
             PPayType str = (PPayType) iterator.next();
             if (str.getPayTypeName().equals("BadPaymentMethod")) {
                 index = i;
             }
             i++;
        }
        if (index != -1) {
             list.remove(index);
        }
        return list;
    }
}
```

7. Build the custom **.jar** file and deploy this customization.
8. Restart Customer Center.

Advanced Customer Center Concepts

This section describes these Customer Center concepts:

- Components Used in Customer Center
- About the BAS Data Flow with a Swing-Compatible UI
- About Field Specifications
- About Controller Processing
- Creating a Self-Signed Java Security Certificate

For information on the Business Application SDK (BAS) framework used by Customer Center and Self-Care Manager, see "Understanding the BRM Business Application SDK Framework".

Components Used in Customer Center

This section describes the main Portal Infranet-aware (PIA) components used in Customer Center, including "Graphical Components", "Nongraphical Components", and "Data Manager Components".

Portal Infranet-Aware Components

The Customer Center framework includes Portal Infranet-aware (PIA) components. These components, or widgets, are extensions to their corresponding Swing components and have references to specific storable class fields. They contain APIs that allow you to map them to specific fields in BRM storable classes, such as **/account**. PIA components have implementations that allow individual components to encapsulate data retrieval and user interface (UI) display.

> **Note:**
>
> Customer Center SDK also includes basic components, such as links, headers, and dates that are useful in accounting and billing applications.

Customer Center consists of panels that display PIA components. These components can be made aware of specific BRM storable class fields. They can perform all the tasks required to retrieve, display, and save BRM data; so when you add a field or change the properties of a panel, you do not have to implement server functionality to communicate with BRM.

The Customer Center framework includes classes for graphical and nongraphical components as well as classes for managing changes made to the components during a login session.

The following sections describe the main components in the framework. For detailed information about the methods and API used, see *Customer Care API Reference*.

Graphical Components

The Java classes in Table 47-3 comprise the main Customer Center graphical components:

Table 47-3 Customer Center Graphical Components

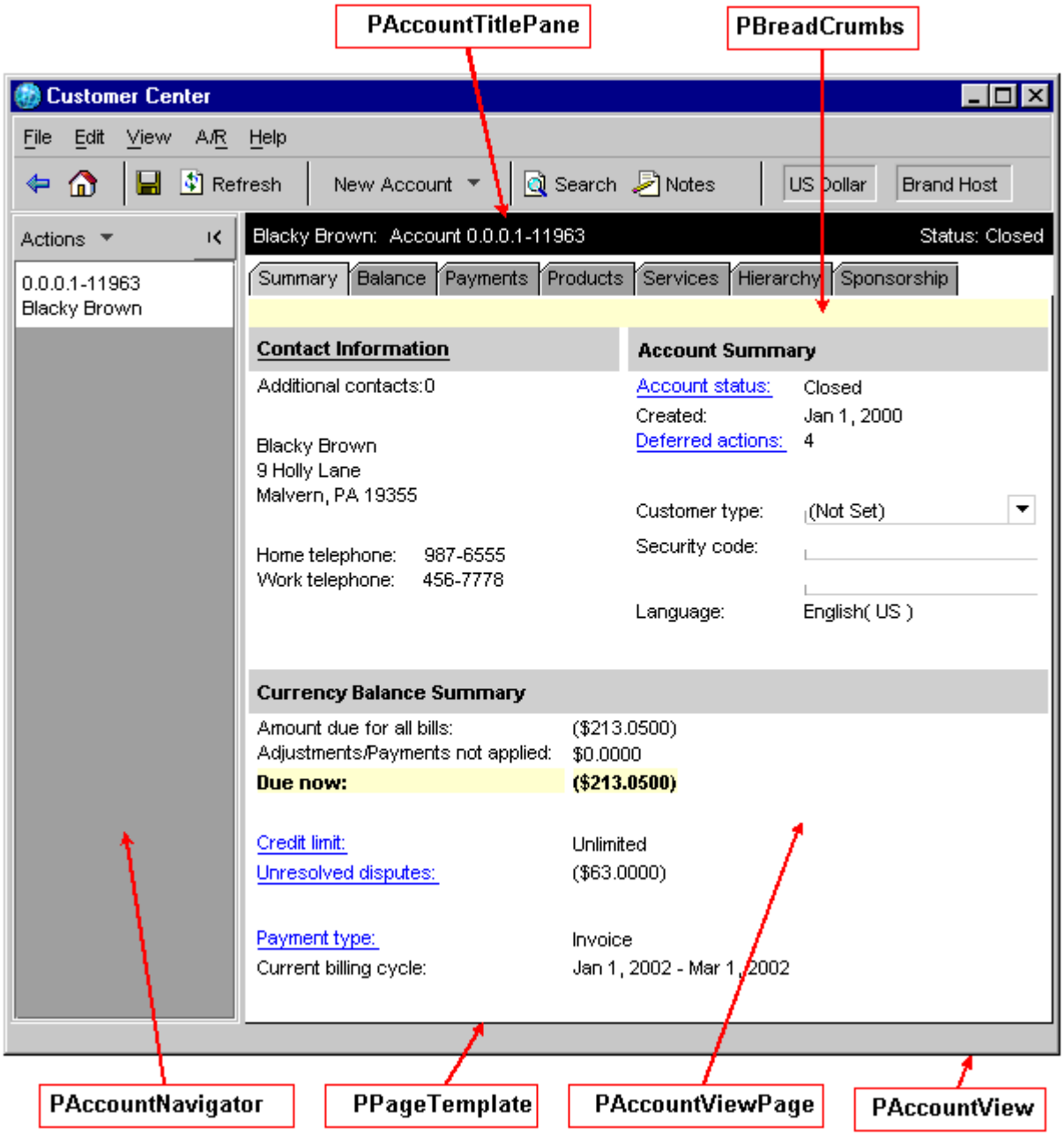
Component	Description
PAccountNavigator	A columnar component that displays a list of open accounts. As the CSR opens accounts, they are added to this graphical list with the most recently opened account at the top. This component also provides the capability of closing accounts and removing them from view in Customer Center.
PBreadCrumbs	Displays a visual history of the various data screens the CSR has visited within a single tab. The history is presented as selectable links. **Note:** Each account is displayed in a tabbed pane component, and each tab can have any number of virtual screens. This means it is possible to stay within the same visual tab and view many different screens.

Table 47-3 (Cont.) Customer Center Graphical Components

Component	Description
PPageTemplate	The main component found on each tab in the account view. It contains the **PBreadCrumbs** component along the top, and a Business Application SDK (BAS) **PIAPanelGroup** component below that. The **PIAPanelGroup** enables the virtual tabs feature of the client by providing a mechanism for embedding a number of screens on top of each other while displaying only a single screen at any one time. The **PPageTemplate** hooks the **PBreadCrumbs** and **PIAPanelGroup** components together. This allows the **PBreadCrumbs** component to determine which screen is displayed in the **PIAPanelGroup**. The **PMaintenancePage** subclass you create exists within the **PPageTemplate** on a given tab.
PAccountView	Provides the concrete implementation of **PAccountViewContext** in Customer Center. This is implemented as a tabbed pane component, with each tab containing a **PPageTemplate** component.
PActiveAccountManager	Responsible for managing the components that display account data. Like the **PPageTemplate** component, it uses a **PIAPanelGroup** to manage the display of **PAccountView** components. Each time a new account is opened in Customer Center, **PActiveAccountManager** delivers a **PAccountView** component that can be used to display the account's data. As the user switches between open accounts, the **PActiveAccountManager** tracks those changes and displays the appropriate **PAccountView**. As accounts are closed in the client, the associated **PAccountView** component is removed from the panel group. Essentially, this component manages a stack of tabbed panes with only the active one visible. It takes up all the space to the right of the **PAccountNavigator**.

Figure 47-1 shows the areas of the interface that some graphical components control:

Figure 47-1 Customer Center Graphical Components



Nongraphical Components

The Java classes in Table 47-4 comprise the main Customer Center nongraphical components:

Table 47-4 Nongraphical Components

Component	Description
PCustomerCenterContext	Provides a context for standalone Javabean components within Customer Center. Components can use this context to: • Open and close accounts • Register for notification when accounts are opened or closed • Access the global data managers: **PBrandManager**, **PCurrencyManager**, **PSaveManager**, and **PrintManager** • Retrieve a model handle to the currently active account or its context. Each open account contains a further set of API encapsulated in **PAccountViewContext**.
PAccountViewContext	A context which abstracts the mechanism used to display existing account data. This context encapsulates the API for enabling drill-downs in the client and accessing the account's **PRefreshManager**. By default, Customer Center uses a tabbed pane component. You can implement the **PAccountViewContext** API to use a different type of component.
PMaintenancePage	A concrete implementation of the base page interfaces used for account maintenance. This component implements **PAccountViewPage** which extends **PCCPage**. If you are building a new page, you should subclass **PMaintenancePage**.
PAccountInfo	A data class instance that exists for every open account in Customer Center. The data can be retrieved from the **PCustomerCenterContext**. This data class contains information about an account. When an account is opened, this data is retrieved and made available for all components. It can be marked for refresh if data contained in the class has been modified and then refreshed from BRM using an API call.

Data Manager Components

The Java classes in Table 47-5 comprise the main Customer Center data manager components:

Table 47-5 Data Manager Components

Component	Description
PBrandManager	Encapsulates the management of and access to the current BRM scope. Used to determine what the current scope is or when the scope changes. This class also contains an API for determining account scope.
PCurrencyManager	Encapsulates management and notification for currency changes. This is useful if the accounts in the system use multiple currencies. Most pages do not need to use this class. If you follow the suggestions for currency toggling described in "Currency Toggling", the framework handles currency toggling automatically.
PSaveManager	Encapsulates functionality for saving data and identifying errors in a page. It also contains the logic for determining whether the CSR should be prompted before data is saved.

Table 47-5 (Cont.) Data Manager Components

Component	Description
PRefreshManager	Encapsulates data synchronization across an account view. Use this class to register for notification of data changes or to broadcast data changes to other registered pages. **Note:** Each account view is recycled when it is closed and a new refresh manager is created each time an account is opened. This means that a page must register or unregister with the refresh manager each time. See the **RefreshTest.java** example and look specifically at the **addNotify()** and **removeNotify()** methods.

About the BAS Data Flow with a Swing-Compatible UI

PModelHandle is a client-side representation of an flist. Because UI code typically does not directly access the PCM library, BAS creates this object for client use. **PModelHandle** is turned into a real flist on the server for use in opcodes. A **PModelHandle**, like an flist, can also represent storable classes. So it is possible to have a **PModelHandle** that represents an account or an event.

Every BAS widget (the PIA widgets) has a **setModelHandle** method. This is how data is passed in to the component. For the container PIA widgets (panel, scroll pane, and tabbed pane) the **setModelHandle** method automatically propagates the **PModelHandle** to all child components. For example, if you have a **PIACustomizablePanel** with ten **PIATextField** components on the panel, one call to **setModelHandle** on the panel is enough to set the **PModelHandle** for all ten **PIATextField** components.

To get data out of the PIA widgets, BAS uses the **collectData** method. Essentially, you create an **UNTYPED PModelHandle** from scratch and hand that to a new **PCollectDataEvent**. This event is passed into **collectData**, which is available on every PIA widget.

Regardless of the type of data collection (you can create many different types of **PCollectDataEvent**), each PIA widget properly updates the **PModelHandle** passed in with the event. For example, to gather all data modified by a user you could issue a **PCollectDataEvent** of type **FOR_DIRTY**. Only the modified PIA widgets would append their data to the **PModelHandle** that is passed in. This would happen automatically by calling **collectData** on the encompassing panel.

About Field Specifications

Each PIA widget contains two methods that bind the widget to a specific field in BRM. These widgets enable the **setModelHandle** and **collectData** methods to work.

- **setDisplayFieldDescription**: This method is a read data mechanism. When **setModelHandle** is called on a widget, the widget internally attempts to extract the BRM field identified by the **setDisplayFieldDescription** specification from the associated **PModelHandle**.

  For example, if a widget is passed a **PModelHandle** that represents an account object and a text field has a display field description of **FldNameinfo[1].FldLastName**, the text field can automatically extract the value of **FldLastName** from the **PModelHandle** and display it.

- **setModelFieldDescription**: This method is a write data mechanism. It performs the reverse function provided by **setDisplayFieldDescription**. This specification indicates how to write a particular piece of data to BRM. It is a **String** representation of what the input flist to an opcode looks like.

In the last name example above, the model field description for storing the last name is identical to the display field description. However, that is a rare case, since most fields in BRM are read from an flist in one format, but passed in an input flist to an opcode in a different format.

About Controller Processing

The previous sections describe how data goes into a panel, and how it is extracted from that panel. If you need to perform additional data processing before your panel receives the data, use the controller for the panel. Each PIA widget contains the method **getControllerClassName**. This identifies the class name of the server-side component (controller) for your panel.

When **setModelHandle** is called on the panel, the **update** method is invoked remotely on the controller. This happens before any widget contained in the panel receives the data. If any type of conversion, lookup, or translation of data needs to occur before passing the data along, this is how to do it.

Building Your Customer Center Customizations

To build your Customer Center customizations, follow these steps:

1. Creating a Self-Signed Java Security Certificate
2. Building Your Customization Files
3. Testing Your Customizations

Creating a Self-Signed Java Security Certificate

If you do not have an authenticated security certificate or do not want to use it for testing your customizations, use the **makecertificate** script to create a self-signed certificate.

> **Note:**
>
> The certificate created by the **makecertificate** script is not a validated, secure certificate. It is not an appropriate certificate to copy to the Web server you use to deploy Customer Center to your CSRs. Its use is limited to internal testing environments.

1. Run the **makecertificate** script, located in the *CCSDK_home*/**CustCntr/bin** directory.
2. Answer the following prompts as they appear, or press ENTER to accept the default entries displayed in brackets:

```
Enter keystore password: yourPassword
What is your first and last name?
   [Unknown]: yourFirstName yourLastName
What is the name of your organizational unit?
   [Unknown]: yourUnitName
What is the name of your organization?
   [Unknown]: yourOrganizationName
What is the name of your City or Locality?
   [Unknown]: yourCity_or_Locality
What is the name of your State or Province?
```

```
   [Unknown]: yourState_or_Province
What is the two-letter country code for this unit?
   [Unknown]: yourCountryCode
```

3. When prompted to verify whether the entries you made are correct, type **yes** or press ENTER to verify or change the entries you made after the keystore password.

   If you accept the default answer, **no**, the script redisplays each of the prompts with your answers listed as the default entry in brackets. You can enter a new value or press ENTER to keep the current values.

4. After you verify that your entries are correct, the script composes a certificate and then prompts you to enter a key password for the test certificate's alias, **customcert**. Enter a different password or press ENTER to reuse the keystore's password.

5. The certificate is stored in a keystore file named **certificate_keystore** in the *CCSDK_home***/bin** directory.

Modifying the signjar Script

Before running the **buildAll** script to build and sign the **ccCustom.jar** file for Customer Center, first edit the *CCSDK_home***/CustomerCareSDK/CustCntr/custom/signjar** script to add your entries for the following values:

- set KEYPASSWORD=*yourPassword*
- set STOREPASSWORD=*yourPassword*

If you created a self-signed certificate with the **makecertificate** script, use the password values you entered to create that certificate.

If you have an authentic Java Security Certificate, use the password values associated with that certificate. Copy your certificate keystore file to the **custom** directory.

Building Your Customization Files

You use the **buildAll** script in the *CCSDK_home***/CustomerCareSDK** directory to build your customized files.

Requirements

Before using the **buildAll** script, verify that:

- The appropriate entries are in the ***.properties** files. See:
  - Using Configurator
  - Modifying the Customer Center Properties Files
- If you created custom source files to extend Customer Center, the source files are in the *CCSDK_home***/CustomerCareSDK/CustCntr/custom** directory.
- Your **certificate_keystore** file is in the *CCSDK_home***/CustomerCareSDK/CustCntr/custom** directory.

  See "Creating a Self-Signed Java Security Certificate".

Using the buildAll Script

To build the **ccCustom.jar** file for Customer Center:

1. Open a command shell and go to *CCSDK_home/***CustomerCareSDK/lib**.

2. Run the **buildAll** script:
   a. To remove class files from any previous custom builds:

      ```
      buildAll CustCntr clean
      ```

   b. To build a new copy of the **ccCustom.jar** file:

      ```
      buildAll CustCntr
      ```

> **Note:**
>
> For more information on the **buildAll** script, see "About Compiling and Packaging Your Customizations" and "Syntax for the buildAll Script".

When the **buildAll** script finishes, you can deploy your Customer Center customizations. See "Deploying Your Customer Center Customizations".

Testing Your Customizations

You can test your customizations before you deploy them by running your local version of Customer Center (the **runCustomerCenter** batch file) in the *CCSDK_home*/**CustomerCareSDK/CustCntr/bin** directory.

> **Note:**
>
> The local version of Customer Center does not require access to the Customer Center Web Start server, but it does require access to a CM.

1. If you are adding custom fields, add **ccCustomFields.jar** to the CLASSPATH as in this example:

   ```
   set CLASSPATH=f:/CCSDK_home/CustCntr/custom/ccCustomFields.jar;%CLASSPATH%
   ```

2. Go to the *CCSDK_home*/**CustomerCareSDK/CustCntr/bin** directory**.**
3. Double-click the **runCustomerCenter** batch file.

   Your local version of Customer Center starts.
4. Log in to Customer Center when prompted.

> **Note:**
>
> You must specify the host and port of a working BRM system when logging in.

5. Verify that your customizations appear and work properly in the Customer Center.
6. Close Customer Center.

> **Note:**
>
> In the "Building Your Customization Files" procedure, you can use **runCustomerCenter** to test your customizations without building **jar** files if:
>
> - You are making customizations with Configurator or by editing the customizes properties files, *and*
> - You are *not* adding new fields, panels, or customized help.

Deploying Your Customer Center Customizations

This section describes how to deploy your Customer Center customizations.

Deploying Customer Center Customizations on Linux

This section describes how to deploy your customizations to the Web Start server.

1. Create a new subdirectory named **custom** in the directory where you installed Customer Center on your Web server.
2. Make a backup copy of the **CustomerCenter_en.jnlp** file on your Web server.
3. Edit the **CustomerCenter_en.jnlp** file in the directory where you installed Customer Center and remove the comments around the following line if they are present:

   ```
   <!-- <extension name="Customized" href="custom/custom.jnlp"/> -->
   ```

4. Copy the **custom.jnlp** and, if present, **ccCustomFields.jar** files from the *CCSDK_home*/**custom** directory to the directory where you installed Customer Center on your Web server.
5. Copy the **ccCustom.jar** and, if present, the **ccCustomHelp.jar** files from the *CCSDK_home*/**custom** directory to the **custom** directory where you installed Customer Center on your Web server.

The next time a CSR accesses the link that opens the **CustomerCenter_en.jnlp** file from your Web server, Web Start installs the updated application to their system.

For installing Java Web Start and Customer Center, see *BRM Installation Guide*.

About the Customer Center Properties Files

Customer Center SDK includes three sets of properties files that define the Customer Center interface:

- Default Properties Files
- Configurator Properties Files
- Customized Properties Files

Parameters in the Configurator properties files take precedence over like parameters in the default properties files. Parameters in the customized properties files take precedence over like parameters in the default and Configurator properties files.

Default Properties Files

The *CCSDK_home*/**CustomerCareSDK/CustCntr/Settings** directory contains copies of properties files shipped with Customer Center. The two files that define the default interface are:

- **CustomerCenter.properties** defines several non-text features of the Customer Center UI, for example:
  - The order of panes in the New Account wizard
  - The Account Maintenance view
  - Which threshold settings to use
  - The home page to use when no accounts are open
  - The search entry page to use
  - The color and font used by the fields
  - The number and sequence of panels in a window or wizard
- **CustomerCenterResources.properties** contains the names of images and the localizable text for the panels, fields, and messages that appear in the UI.

These properties files are shipped in signed **jar** files that are installed with Customer Center and can't be directly modified.

> **Note:**
>
> Do not modify these or any other properties files under the **CustCntr/Settings** directory. Customer Center uses these in **jar** files that are signed with an authentic security certificate from BRM. If you directly modify these files and attempt to include them in a custom **jar** file signed with a different certificate, Web Start returns an error. To change a property or string, copy it to the **Customized.properties** or **CustomizedResources.properties** files.

Configurator Properties Files

When you save changes you make in Configurator, the parameters are saved in the **WizardCustomizations.properties** and **WizardCustomizationsResources.properties** properties files in the *BRM_home*/**CustomerCareSDK/CustCntr/bin** directory, where *BRM_home* is the directory in which the BRM server software is installed.

Parameters in the Configurator properties files take precedence over like parameters in the default properties files.

> **Note:**
>
> Do not directly modify or add entries to the **WizardCustomizations** files. These files are rewritten every time you save Configurator changes.

Customized Properties Files

The *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory contains the following files for overriding the default properties defined in the installed Customer Center properties files:

- **Customized.properties** modifies the appearance of the UI components in Customer Center.
- **CustomizedResources.properties** modifies the names of images and localizable text that appear in Customer Center.

Use these properties files to make global changes that affect the whole application or several related fields in an application.

The parameter values in the customized properties files take precedence over like parameter values in the default and Configurator properties files.

For examples of customizations you can make by modifying the customized properties files, see "Modifying the Customer Center Properties Files".

For more information on the properties files parameters, see the comments in the "Default Properties Files".

> **Tip:**
>
> You can copy and paste default parameters from the "Default Properties Files" into their "Customized Properties Files" counterparts and change the parameter values as required.

Other Properties Files

Customer Center SDK includes these additional properties files in the *CCSDK_home*/**CustomerCareSDK/CustCntr/Settings** directory:

- **CCViewResources.properties** contains additional strings used in Customer Center, such as warning messages.
- **GSMManagerResources.properties** contains the names of images and the localizable text for GSM Manager panels that display extended service information. This property file is only valid when the GSM Manager Customer Center Extension is installed.

> **Note:**
>
> Do not change entries directly in these resource files. Instead, copy the entry to the **CustomizedResources.properties** file and change the parameter as required. Parameters found in the **CustomizedResources.properties** file takes precedence over the value in the same parameter in the **CCViewResources.properties** and **GSMManagerResources.properties** files.

The *CCSDK_home*/**CustomerCareSDK/CustCntr/bin/Infranet.properties** file provides specific server settings for starting the Configurator application or for using the **runCustomerCenter** script to start a standalone Customer Center session.

Deploying Customer Center Customizations on Windows

This section describes how to deploy your customizations to the TomCat server.

1. Copy the certificate keystore file (by default at *CCSDK_home*/**CustomerCareSDK/CustCntr/bin**) to **C:\Program Files\Apache Group\Tomcat 4.1\webapps\ROOT\customercenter**.
2. Copy the **signjar.bat** file from *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** to **C:\Program Files\Apache Group\Tomcat 4.1\webapps\ROOT\customercenter\lib**.
3. Edit the **signjar.bat** file and enter the password:

   ```
   set KEYPASSWORD= password
   set STOREPASSWORD= password
   where password refers to the keystore password
   ```

4. Go to **C:\Program Files\Apache Group\Tomcat 4.1\webapps\ROOT\customercenter\lib**.
5. Using an application such as WinZip, open all the **.jar** files and remove the ORACLE_C.RSA and ORACLE_C.SF files from the archive.
6. Copy the **ccCustomjar** file from *CCSDK_home* **\CustomerCareSDK\lib** to **C:\Program Files\Apache Group\Tomcat 4.1\Webapps\ROOT\customercenter\lib**.
7. Go to **C:\Program Files\Apache Group\Tomcat 4.1\webapps\ROOT\customercenter\lib**.
8. Run the following command, which signs all the JAR files:

   ```
   signjarbat jarFileName
   ```

9. Open the *CCSDK_home*/**CustCntr/custom/CustomerCenter_en.jnlp file** and add the path to the **ccCustomjar** file in the **Resources** area. Replace **<-NEWJAR-/>** with the following:

   ```
   <jar href="lib/ccCustomjar.jar"/>
   ```

Customer Center Customization Examples

When you install Customer Center SDK, separate folders with the appropriate source and support files provide examples for extending Customer Center and Self-Care Manager. Many of the example folders include brief README files to explain the purpose of each example.

You can run many of the examples in place by using the **testExamples.bat** script.

Table 47-6 describes the Customer Center extension examples in the *CCSDK_home*/**CustomerCenterSDK/CustCntrExcamples** directory. Read the **readme.txt** files and the comments in the source files of each example for further information on their functionality and how to use them for creating your own customizations.

Table 47-6 Customer Center Extension Examples

Directory	Contents
AccountCreation	**AcctCreationTapNonGUI.jav**a, an example that demonstrates how you might use the New Account wizard without adding a panel to the UI. **CredScoreProfile.java** is an example of tapping into the New Account wizard with a custom page. A panel for a new profile subclass is added, which simulates the retrieval of a credit score that determines which plans are available for purchase. **CreditScoreProfile.sce**, a BRM storable class definition file, which defines a sample object used by the examples in this directory. To see the contents of this file, use Storable Class Editor. **Customized.properties**, a collection of properties required for implementing these examples in the Customer Center UI. To implement customizations, you copy and paste these entries into the **Customized.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory. **CustomizedResources.properties,** a collection of resource properties required for implementing these examples in the Customer Center UI. To implement customizations, you copy and paste these entries into the **CustomizedResources.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory. **DepositSimulator.java** provides a simulated deposit requirement, represented as a set of checkboxes for a CSR to select. **ManageFinishButton.java**, an example that demonstrates how you can control the **Finish** button in the New Account wizard.
Controllers	**ControllerTest.java**, an example of invoking methods on your controller. The controller for this class contains two methods that retrieve the number of service objects and profile objects associated with the current account. **ControllerTestBeanImpl**, an example of a controller that can be used in Customer Center.
General	**Customized.properties**, a collection of properties required for implementing these examples in the Customer Center UI. To implement customizations, you copy and paste these entries into the **Customized.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory. **CustomizedResources.properties**, a collection of resource properties required for implementing these examples in the Customer Center UI. To implement customizations, you copy and paste these entries into the **CustomizedResources.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory. **DrillDownTest.java** contains a number of examples demonstrating how to perform drill downs using the Customer Center framework. **PermissionTest.java** contains examples of how to retrieve CSR permissions. **RefreshTest.java** contains examples of registering for refresh events. **SaveTest.java** contains an example of saving data to BRM. It also demonstrates how to use the PIA widget API and invoke a controller other than the default controller created for this panel.
HomePage	**CustomHomePage**, an example of a branded Customer Center home page. **Customized.properties**, a collection of properties required for implementing this example in the Customer Center UI. To implement customizations, you copy and paste these entries into the **Customized.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory. **LaunchBrowserPage**, a sample home page that demonstrates using the JNLP API to invoke the system browser.

Table 47-6 (Cont.) Customer Center Extension Examples

Directory	Contents
Notes	**Customized.properties**, a collection of properties required for implementing this example in the Customer Center UI. You copy and paste these entries into the **Customized.properties** file in *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory before running the **buildAll** script. **HTMLNotes.java** is an example of an alternate notes display.
Profile	**CDProf.java**, a profile panel example that can be used during account creation and account maintenance. A very simple example that captures the birthday and gender of the account holder. **CDProf.sce**, a BRM storable class definition file, which defines a sample object. This sample object is used by the CDprof example in this directory. To see the contents of this file, use Storable Class Editor. **CombinedProfPage.java**, an example of profile panels for multiple **/profile** subclasses that are incorporated into one UI panel. **CSProf.java**, a profile panel example that can be used during account creation and account maintenance. This example captures the credit score. **CSProf.sce**, a BRM storable class definition file, which defines a sample object. This sample object is used by the CSProf example in this directory. To see the contents of this file, use Storable Class Editor. **ProfileTemplate.txt**, a template for an empty Profile panel. Use this template as a starting point for creating a profile panel for use with the New Account wizard and for account maintenance.
Events	An example event displayer. You can use this generic code to retrieve and display most BRM events. This code also demonstrates how to export the displayed date to HTML.
Service	**Customized.properties**, a collection of properties required for implementing this example in the Customer Center UI. You copy and paste these entries into the **Customized.properties** file in the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory before running the **buildAll** script. **PEmailPanel.java**, an example of a service panel. The service object used for this example is **/service/email**. **ServiceTemplate.txt**, a template for an empty extended-service panel. Use this template as a starting point for creating a panel for an extended service. The **PGPRSPanel**, **PGSMPanel**, **PMSExchangeOrgPanel**, and **PMSExchangeUserPanel** classes are additional examples of extended service panels.
Summary	**AddToSummaryPage.java**, an example of embedding panels directly into the summary page. **Customized.properties**, a collection of properties required for implementing these examples in the Customer Center UI. You copy and paste these entries into the **Customized.properties** file in the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory before running the **buildAll** script. **CustomizedResources.properties**, a collection of resource properties required for implementing these examples in the Customer Center UI. You copy and paste these entries into the **CustomizedResources.properties** file in the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory before running the **buildAll** script. **WrappedSummaryPage.java** is an example of wrapping the existing Customer Center summary page with additional information or fields at the bottom of the page.

48

Using Configurator to Configure Customer Center

Learn about the Oracle Communications Billing and Revenue Management (BRM) Configurator application included with Customer Center Software Development Kit (SDK).

Topics in this document:

- About Configurator
- Using Configurator
- Configuring Customer Center Account Maintenance Pages
- Configuring the Customer Center New Accounts Wizard
- Using the Configurator Resource String Editor
- Additional Configured Profile Panel Examples

About Configurator

You use Configurator to configure the most commonly modified features of Customer Center. The Configurator interface mimics the tab organization of the Customer Center interface, so you can easily locate the features you want to modify.

When you choose **File - Save** in Configurator, Configurator saves the changes to the **WizardCustomizations.properties** and **WizardCustomizationsResources.properties** configuration files in the *BRM_home*/**CustomerCareSDK/CustCntr/bin** directory, where *BRM_home* is the directory in which the BRM server software is installed.

For an overview of all Customer Center properties files, see "Modifying the Customer Center Properties Files".

Using Configurator

To use Configurator to configure Customer Center:

1. Go to the *CCSDK_home*/**CustomerCareSDK/CustCntr/bin** directory.
2. Double-click the **runConfigurator** executable script.

   The Login dialog box appears.
3. Enter your login and password.
4. Click the **Connection Info** button to display and, if needed, change the values for **Host** and **Port**. These entries are automatically filled in with the values you provided when you installed Customer Center SDK.

> **Note:**
>
> You can enter values that point to a different BRM server or database after installing Customer Center SDK.

5. Click **OK**.

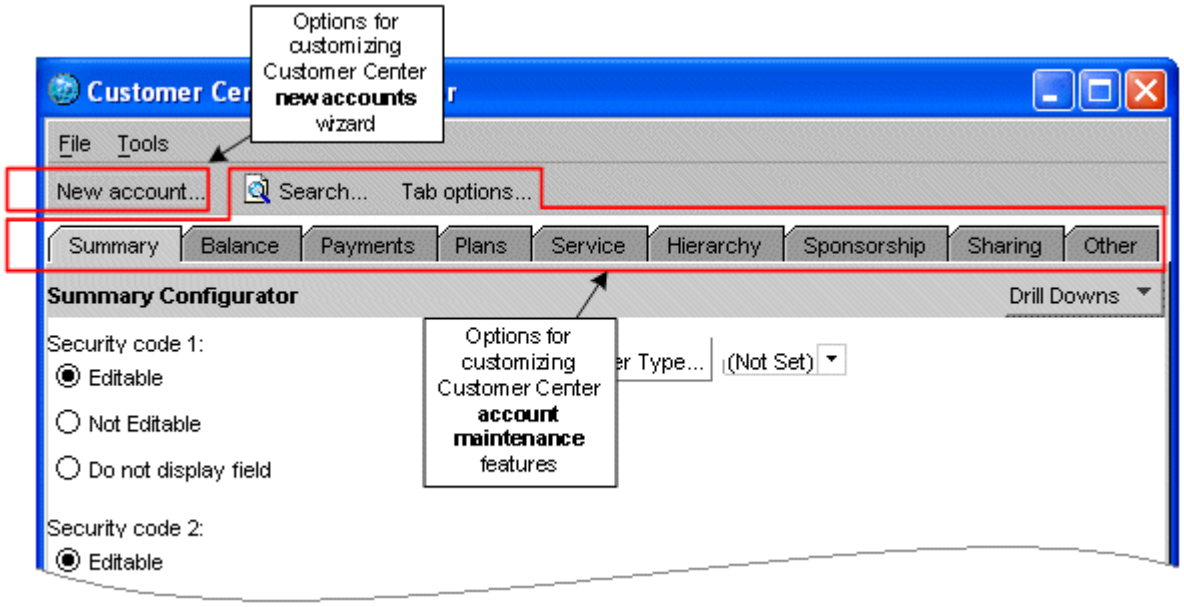
   The Configurator application opens with the **Summary Configurator** tab on top.

6. Make your configurations:

   - To configure Customer Center panels that handle changes to existing accounts, see "Configuring Customer Center Account Maintenance Pages".
   - To configure Customer Center panels in the New accounts wizard, see "Configuring the Customer Center New Accounts Wizard".

   Figure 48-1 shows where to find these options in the initial Configurator panel:

Figure 48-1 Customer Center Configurator Panel



7. Save your changes.

What's Next?

Continue coding your configurations using other methods described in "Customizing and Configuring Customer Center", or proceed to "Building Your Customer Center Customizations".

Configuring Customer Center Account Maintenance Pages

This section describes how to configure fields and pages used in Customer Center to display information about existing accounts.

Using the Account Maintenance Configurator Tabs

The tabs and toolbar selections in Configurator mirror the ones found in Customer Center.

Table 48-1 lists the Configurator tab selection options for configuring the corresponding account maintenance items in Customer Center.

Table 48-1 Configurator Account Maintenance Configurator Tab Selections

Configurator Account Maintenance Tab	Tab Description
Summary Configurator	Configures options in the **Customer Center Summary** tab
Contacts Configurator	Configures options in the Customer Center Contacts drill-down area of the **Summary** tab
Balance Configurator	Configures options in the **Customer Center Balance** tab
Payment Configurator	Configures options in the **Customer Center Payment** tab
Plan Configurator	Configures options in the **Customer Center Plans** tab
Service Configurator	Configures options in the **Customer Center Services** tab
Hierarchy Configurator	Configures options in the **Customer Center Hierarchy** tab
Sponsorship Configurator	Configures options in the **Customer Center Sponsorship** tab
Sharing Configurator	Configures options in the **Customer Center Sharing** tab
Other Settings	Handles configuration options that are not specific to a particular account maintenance tab

Table 48-2 shows the Configurator toolbar selection options for configuring the corresponding account maintenance items in Customer Center.

Table 48-2 Configurator Account Maintenance Toolbar Selections

Account Maintenance Toolbar Selection	Tab Description
Account Search Results Configurator	Configures the Customer Center account Search panel
Tab Options	Configures the Customer Center tabbed pages

Summary Configurator

Click the **Summary** tab to configure the fields and choices displayed on the Customer Center Summary page.

The **Drill Downs** menu on this tab displays two choices:

- Summary
- Contacts

Choose **Summary** to open the Summary Configurator, which displays the choices listed in Table 48-3:

Table 48-3 Summary Tab Items

Item	Summary Page Action	Default
Security code 1	Select an option to set permissions for the first **Security code** field. • **Editable** • **Not editable** • **Do not display field**	**Editable**
Security code 2	Select an option to set permissions for the second **Security code** field. • **Editable** • **Not editable** • **Do not display field**	**Editable**
Deferred actions	Select or clear the option to display the **Deferred actions** field.	**Displayed**
Language (locale)	Select or clear the option to display the **Language** field.	**Displayed**
Dispute	Select or clear the option to display these dispute options: • **Unresolved Dispute** link on the Summary and Balance pages • **Dispute** option in the **Type** drop-down list in the Search dialog box on the A/R Actions On All Bills page • **Open Dispute** and **Settle Dispute** options on the Bill Details page • **Open Dispute** and **Settle Dispute** options on the **Item Changes Action** menu	**Displayed**
Balance summary section	Select or clear the option to display: • The Currency Balance Summary pane on the Summary page • The Balance Summary pane on the Balance page.	**Displayed**
Payment type	Select or clear the option to display the Billing Payment Method section of the **Payments** tab. **Note:** This option only affects the Payment page. It does not affect the Summary page.	**Displayed**
Show service in change status list	Select or clear the option to display or hide the services in the **Account/service** drop-down list in the Change Account/Service Status panel in the **Summary** tab.	**Displayed**
Modify Customer Type	Click Modify Customer Type to change the list of customer type options. See "Modifying the Customer Type List".	**Default list:** • Bronze • Silver • Gold • Platinum

Modifying the Customer Type List

The **Summary** tab in Customer Center Configurator includes a **Modify Customer Type** button that allows you to modify the customer type pull-down list.

Adding a customer type

To add a customer type to the selector pull-down list:

1. In the **Summary** tab, click **Modify Customer Type**.

   The Modify Customer Type window appears.
2. Type a BRM value in the **Map the Infranet value** field.
3. Type a string to associate with the BRM value in the **To the string** field.

> **Note:**
>
> The text in the **To the string** field will appear in the **Customer Type selector** pull-down list.

4. Click **Add**.

Deleting a customer type

1. In the **Summary** tab, click **Modify Customer Type**.
2. Click the customer type you want to delete.
3. Click **Delete**.

Contacts Configurator

Choose **Contacts** from the **Drill Downs** menu to open the Contacts Configurator.

The Contacts Configurator options shown in the following table affect the text entry fields displayed in the **Contact Information** drill down on the **Summary** page. If you remove a field from the display, the fields that follow it are moved up on the displayed pages.

To change only the fields displayed in the New Account wizard, click the **New account** toolbar button and access the Contact Configurator for the "Configuring the Customer Center New Accounts Wizard". Set the values for the items listed in Table 48-4.

> **Note:**
>
> The **Last name**, **Address**, and **Phone** fields cannot be removed.

Table 48-4 Contacts Configurator Items

Item	Action	Default
Company	Select or clear the option to display the **Company** field.	**Displayed**
Job Title	Select or clear the option to display the **Job title** field.	**Displayed**
Salutation	Select or clear the option to display the **Salutation** field.	**Displayed**
First Name	Select or clear the option to display the **First name** field.	**Displayed**
Middle Name	Select or clear the option to display the **Middle name** field.	**Displayed**
City	Select or clear the option to display the **City** field.	**Displayed**
State	Select or clear the option to display the **State/Province** field.	**Displayed**
Zip	Select or clear the option to display the **ZIP/Postal** field.	**Displayed**

Table 48-4 (Cont.) Contacts Configurator Items

Item	Action	Default
Country	Select or clear the option to display the **Country** field.	**Displayed**
E-mail	Select or clear the option to display the **Email** field.	**Displayed**

Balance Configurator

Click the **Balance** tab to open the Balance Configurator. Use the Balance Configurator to configure the options (shown in Table 48-5) in the **Balance** tab of Customer Center:

Table 48-5 Balance Configurator

Item	Balance Panel Action	Default
Event browser	Select or clear the option to display the **Event Browser** menu choice on the **Edit** menu. **Note:** This choice doesn't affect the **Event Adjustment** action.	**Displayed**
Refund	Select or clear the option to display all of the following: • **Refund Account** item on the **A/R** menu. • **Refund Bill** item on the **Bills - Action** menu.	**Displayed**
Dispute	Select or clear the option to display all of the following: • **Unresolved Disputes** link on the Summary and Balance pages. • **Dispute** choice in the **Type** drop-down list in the **Search – A/R Actions** dialog box on the **A/R Actions On All Bills** page. • **Open Dispute** and **Settle Dispute Action** menu selections in the **Item Charges** panel on the **Bill Details** drill-down page.	**Displayed**
Write-off	Select or clear the option to display the of the following: • **Write Off Account** menu option on the **A/R** menu. • **Write Off Bill** item in the **Bills** section on the **Balance** page. • **Write Off Bill** item in the **Item Charges - Action** menu **o**n the **Bill Details** drill-down page.	**Displayed**
Make "Include Child Amount" checkbox read only	Click the checkbox to enable or disable changing the selection of the **Include Child Accounts** checkbox.	The **Include Child Accounts** checkbox choice can be modified.
Balance summary section	Select or clear the option to display the Balance Summary pane on the Balance page and the Currency Balance Summary pane on the Summary page.	**Displayed**
Noncurrency section	Select or clear the option to display the Noncurrency panel.	**Displayed**
Account adjustment	Select or clear the option to display the **Account Adjustment** link on the Balance and Bill Details pages.	**Displayed**
Item adjustment	Select or clear the option to display the **Item Adjustment** choice in the **Action** menu in the Item Charges panel on the Bill Details drill-down page.	**Displayed**

Table 48-5 (Cont.) Balance Configurator

Item	Balance Panel Action	Default
Event adjustment	Select or clear the option to display the **Event Adjustment** choice in the **Action** menu in the following: • Bills panel on the Balance page • Item Charges panel on the Bill Details page	**Displayed**
Disable View Invoice Button in Bill Details page	Select or clear the option to disable the **View Invoice** button in the **Bills** panel of the **Balance** tab. By default, this option is not selected.	**Enabled**
Tax treatment	Select the option to control whether the customer center representative (CSR) can choose the **Tax treatment** on the adjustment, dispute, and settlement dialog boxes: • **Include tax**: Always perform a tax reversal for the adjustment, dispute, or settlement. If you select this option, the Customer Center dialog boxes do not include tax treatment checkboxes. • **Exclude tax**: Never perform a tax reversal for the adjustment, dispute, or settlement. If you select this option, the Customer Center dialog boxes do not include tax treatment check boxes. • **None**: Allow the CSR to choose whether to include taxes. If you select this option, the Customer Center dialog boxes include tax treatment check boxes. • This is the default setting for Customer Center. While it provides flexibility by letting the CSR make decisions based on the circumstances surrounding the adjustment, dispute, or settlement, it can result in inconsistent application of tax reversals for disputes and settlements. For more information on tax treatment, see "Configuring the Default Tax Treatment for Customer Center " in *BRM Calculating Taxes*.	**None**

Payment Configurator

Click the **Payments** tab to open the Payment Configurator. Use the Payment Configurator to configure the options (shown in Table 48-6) in the **Payment** tab of Customer Center:

Table 48-6 Payment Configurator Items

Item	Payments Panel Action	Default
Payment section	Select or clear the option to display the Payment Setup panel.	**Displayed**
Tax setup section	Select or clear the option to display the Tax Setup panel.	**Displayed**
Credit Card Number - maximum digits allowed	Enter an integer to indicate the maximum number of digits allowed for the **Credit Card Number** field.	**16**

Table 48-6 (Cont.) Payment Configurator Items

Item	Payments Panel Action	Default
CVV2 Number - maximum digits allowed	Enter an integer to indicate the maximum number of digits allowed for the **CVV2** field. **Important:** If you change the default value, you must also customize the PCM_OP_CUST_POL_VALID_PAYINFO policy opcode to validate the number of CVV2 digits entered. See "Specifying the Maximum Number of Digits Allowed for CVV2 Verification" in *BRM Configuring and Collecting Payments*.	**3**
Account Number - maximum digits allowed	Enter an integer to indicate the maximum number of digits allowed for the **Account Number** field for direct debit accounts.	**26**
Bank Number - maximum digits allowed	Enter an integer to indicate the maximum number of digits allowed for the **Bank Number** field for direct debit accounts.	**26**
Custom billing cycle/tax setup class	Enter the custom **PBillingCycleAndTaxSetupPage** class name. For more information, see "Configuring Values in the Billing Day of Month Combo Box".	**Not customized**

Plan Configurator

Click the **Plans** tab to open the Plan Configurator. Use Plan Configurator to configure the options (listed in Table 48-7) in the **Plans** tab of Customer Center:

Table 48-7 Plan Configurator Items

Item	Plan Panel Action	Default
Product History	Select or clear the option to display the **Product History** option in the **Actions** menu on the Plans page.	**Displayed**
Deal History	Select or clear the option to display the **Deal History** option in the **Actions** menu on the Plans page.	**Displayed**
Service History	Select or clear the option to display the **Service History** option in the **Actions** menu on the Plans page.	**Displayed**
Product status	Globally modifies *all* Customer Center user permissions for altering Product Status settings. **Note:** To modify a *specific* user's permissions for altering Product Status settings, see the permission strings displayed when you click **Additional note** under the field. The permission string is entered in the Customer Center Permission dialog box.	**Read and write permissions**

Service Configurator

Click the **Service** tab to open the Service Configurator.

The **Drill Downs** menu on this tab displays two choices:

- **Service**
- **Deferred Actions**

Choose **Drill Downs** - **Service** to open the Service Configurator. Use Service Configurator to configure the options (shown in Table 48-8) on the **Services** tab of Customer Center:

Table 48-8 Service Configurator

Item	Service Panel Action	Default
Allow purchase products	Select or clear the option to display the **Purchase** option.	**Displayed**
Service History	Select or clear the option to display the **Service History** option.	**Displayed**
Status column	Select an option to set permissions for entries in the **Status** column: • **Editable** • **Not editable** • **Do not display field**	**Editable**
Defer action column	Select an option to set permissions for entries in the **Defer Action** column: • **Editable** • **Not editable** • **Do not display field**	**Editable**
Custom SIM Panel Class	Enter the subclass of the **SIMPanel.java** class that you created for charging for Subscriber Identity Module (SIM) changes. For more information, see "Adding Charges for SIM and MSISDN Changes".	**Not customized**
Custom Number Panel Class	Enter the subclass of the **NUMPanel.java** that you created for charging for MSISDN changes. For more information, see "Adding Charges for SIM and MSISDN Changes".	**Not customized**
Disable change in MSISDN number	Select or clear the option to disable changes to the Mobile Station International Subscriber Directory Number (MSISDN) number.	**Enabled**

Choose **Drill Downs** - **Deferred Actions** to open the Deferred Actions Configurator. Set the items shown in Table 48-9.

Table 48-9 Deferred Actions Configurator

Item	Action	Default
Delete deferred actions	Select or clear the option to display the **Delete** button.	**Displayed**
Execute deferred actions	Select or clear the option to display the **Execute Now** button.	**Displayed**

Hierarchy Configurator

Click the **Hierarchy** tab to open the Hierarchy Configurator. Use Hierarchy Configurator to configure the options (shown in Table 48-10) on the **Hierarchy** tab in Customer Center:

Table 48-10 Hierarchy Configurator

Item	Hierarchy Panel Action	Default
Allow accounts to be moved	Select or clear the option to display the **Move** button.	**Displayed**
Allow moves to be deferred	Select or clear the option to display the **Defer the action until** field in the Move – Options dialog box.	**Displayed**
Custom Hierarchy Move Page Class	Enter the name of your extended **com.portal.app.cc.PHierarchyMovePage.java** class for your custom search dialog box. For more information, see "Creating Customized Search Dialogs and Disabling the To Field".	**Not customized**
Allow accounts to be removed	Select or clear the option to display the **Actions – Remove from Hierarchy** option.	**Displayed**
Allow removal to be deferred	Select or clear the option to display the **Action – Remove from Hierarchy – Effective Move Date** option**.**	**Displayed**
Allow the bill in progress to be transferred or carried along	Select or clear the option to display the: • **Transfer the current bill in progress to this account** option in the **Actions – Remove from Hierarchy – Bill in Progress Options** menu • **Transfer the bill in progress to the new parent** option in the **Move – Options – Bill in Progress Options** menu Click the option under Default to set **Transfer the (current) bill in progress** as the default choice for both menus.	**Allowed**
Allow the bill in progress to be billed immediately	Select or clear the option to display the **Bill now** option in the: • **Actions – Remove from Hierarchy – Bill in Progress Options** menu • **Move – Options – Bill in Progress Options** menu Click the option under Default to set **Bill now** as the default choice for both menus.	**Not allowed**
The confirmation dialog's default button is	Select one of these options as the default confirmation button: • **Yes** • **No**	**Yes** **Note:** A CSR can still click the **Cancel** button.
Custom NoHierarchy Page Class	Enter your custom NoHierarchy page class. For more information, see "Adding a Custom NoHierarchy Page".	**Not customized**
Double click account opens the account	Select or clear the option to enable opening any account in a hierarchy by double-clicking it. **Note:** If this feature is disabled, double clicking an account expands or collapses it.	**Enabled**
Show expand/collapse control	Select or clear to display the expand and collapse controls (+ and - icons) in an account hierarchy.	**Not shown**
Expand entire hierarchy by default	If the **Show expand/collapse control option** is selected, select or clear to specify whether the default tree display mode is expanded or collapsed.	**Not expanded**

Table 48-10 (Cont.) Hierarchy Configurator

Item	Hierarchy Panel Action	Default
Expand by default only when total number of accounts are less than	If the **Expand entire hierarchy by default,** enter the threshold number of accounts above which the default display is collapsed. To show the entire hierarchy for any size tree, set the field to **0**. **Important:** Customer Center performance might be affected if this field is set to **0** or a high number. If the number of accounts in the hierarchy is more than the expansion threshold, the account hierarchy is displayed as a collapsed tree and a message window indicates the complete hierarchy cannot be shown.	**Not specified** (a default value of **25** is assumed)

Sponsorship Configurator

Click the **Sponsorship** tab to open the Sponsorship Configurator. Click the **Include sponsorship functionality** checkbox to display or hide sponsorship information from all areas of Customer Center. The default behavior is to display sponsorship information.

Sharing Configurator

Click the **Sharing** tab to open the Sharing Configurator. Use Sharing Configurator to control how Customer Center adds members to sharing groups and to configure the options (shown in Table 48-11) on the **Sharing** tab in Customer Center:

Table 48-11 Sharing Configurator

Item	Sharing Panel Action	Default
Automatically participate in the membership	Select the option to control whether members can automatically participate in group sharing: • **Accept** • **Decline** Selecting **Accept** lets members benefit from sharing groups without having to explicitly join the group by clicking **Participate in Membership**. BRM automatically adds ordered balance groups to the database for all members selected on the Add Members dialog box. For more information, see "Creating or Modifying Multiple Ordered Balance Groups Simultaneously " in *BRM Managing Customers*.	**Decline**

Table 48-11 (Cont.) Sharing Configurator

Item	Sharing Panel Action	Default
Accepting the membership as	Select the option to determine whether new sharing groups are added at the beginning or end of the ordered balance group list for participating services: • **First Priority** • **Last Priority** This setting controls the sequence in which BRM applies discount and charge sharing for a service. This setting also controls how the sharing groups for a service are arranged on the **Sharing** tab and Participate in Membership dialog box. For information on ordered balance groups, see "About Ordered Balance Groups" in *BRM Managing Customers*.	**Last Priority**
Bill unit for group owner	Select the option to display or hide **Bill to put charges in** and **Payment method** fields: • **Show it** • **Hide it** The **Bill to put charges in** field appears on the Charge Sharing Group and Discount Sharing Group dialog boxes. The **Payment method** field appears only on the Charge Sharing Group dialog box.	**Show it**
Ordering the Sharing combo box	Change the order of the sharing groups in the **View** drop-down on the **Sharing** tab. To do so, select a group and click: • **Raise Order** • **Lower Order** The list of groups always include **PDiscount** and **PCharge**. It also includes any new sharing group types you create and add to Customer Center. For information on adding new panels to Customer Center, see "Customizing the Customer Center Interface". For information on adding new sharing types to the **View** drop-down on the **Sharing** tab, see "Customizing Fields in the Sharing Tab".	Discount sharing (**PDiscount**) is first

Other Settings

Click the **Other** tab to configure default behavior for the items in Table 48-12:

Table 48-12 Other Items

Item	Other Action	Default
Display the connection info in the login dialog	Select or clear the option to display the **Connection Info** button in the Login dialog box.	**Displayed** **Tip**: If you are working in a single BRM environment, hide the Connection Info button.
Background image in home page	Enter the package path and name for an alternate image to display in the Customer Center home page.	**com/portal/app/cc/homebg.jpg**

Table 48-12 (Cont.) Other Items

Item	Other Action	Default
Allow an account to have only one plan	Select or clear this option to restrict accounts to have only one plan. When this option is selected, a CSR doesn't have the standard set of options in the **Purchase Options** field in the **Plans - Purchase** window. Instead, the CSR can only select the **Upgrade from** option to transfer the account to a new plan.	**Not restricted** (accounts can have two or more plans)
Enable plan options page when using plans with only optional deals	Enable or disable the **Plan Options** tab in the Account purchase wizard when the plan being purchased has no required deals.	**Not enabled**
Class name for loading the Custom Properties	Enter the class name that implements the **LoadCustomProperties** interface. **Important:** You must define the fully qualified class name, for example, **com.helloworld.MyInterface**. For more information, see "Configuring Dynamic Drop-Down Lists".	**Not customized**
Maximum number of contacts allowed for an account	Enter the maximum number of contacts allowed for an account.	**-1** (No limit to the number of contacts in an account)
Enforce deal customization	Select or clear this option. When selected, Customer Center automatically reminds CSRs to visit the **Customize Products** page if they select a deal with required customization.	Not selected

Account Search Results Configurator

The Account Search Configurator wizard in the Customer Center SDK Configurator includes a Search Criteria section. You use this area to add, change, and delete custom search fields that CSRs use to search for accounts in Customer Center.

> **Note:**
>
> The Customer Center SDK includes sample configurations.

To change Customer Search criteria field options in Customer Center by using the Account Search Configurator, follow these steps:

1. Starting Account Search Configurator
2. Adding a New Search Criteria field
3. Modifying a Search Criteria Field
4. Deleting a Custom Search Criteria Field

Starting Account Search Configurator

1. Start Configurator.

2. Click the **Search** toolbar button to open the Account Search Configurator dialog box. Use this dialog box to:
   - View and modify the list of fields available in the Customer Center Search dialog by default.
   - Add more fields.

Adding a New Search Criteria field

To add a new search criteria field:

1. In the Search Criteria section of the Account Search Configurator Wizard, click **Add New**.
2. In the Attributes window, type the values for the basic attributes fields:
   a. **Property file identifier:** Type a keyword that is unique to the field.

   > **Tip:**
   >
   > For easy identification, use a value that resembles the label name. For example, if you use **Payment type** for the label, you could use **Payment_type** for the **Property file identifier**.

   This key is stored in the **Customized.properties** file when you exit Configurator.

   b. **Label:** The label that appears in the Search dialog box for this field.

   c. (Optional) **Mnemonic:** A single letter that allows the customer service representative (CSR) to access this field quickly.

   For example, if you specify **t** as the mnemonic for the Payment Type field, the CSR can access this field by typing **Ctrl+t**.
3. (Optional) To require CSRs to select from a set of specific values for the field:
   a. Click the **Define values for field** option.

   b. Type a selectable value in the **Define value** field.

   c. Click **Add**.

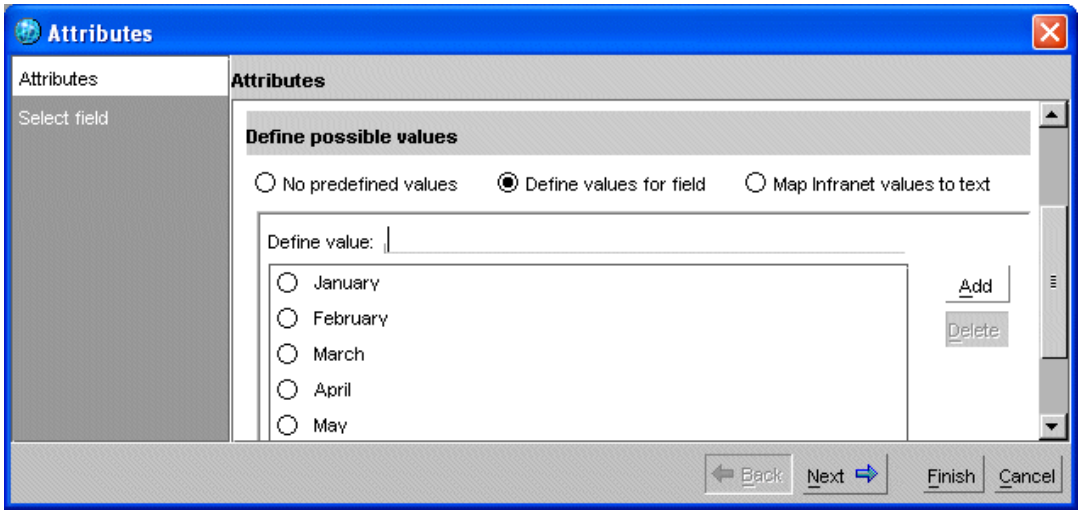
   d. Repeat steps 3b and 3c until all predefined options are added.

   e. (Optional) To specify a default value for the field, select the button to the left of the field.

   f. To delete a field value option, select its row and click **Delete**.

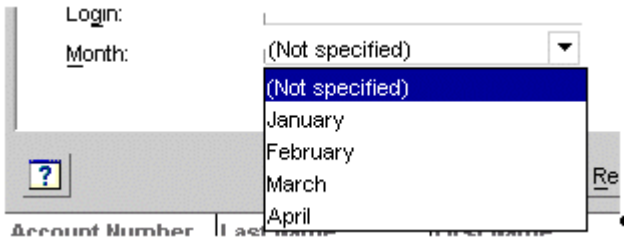
   Figure 48-2 shows the **Month** field added with selectable values:

Figure 48-2 Month Field Attribute Configuration



When a CSR searches on the **Month** field, the drop-down list of months is presented (as shown in Figure 48-3):

Figure 48-3 Customer Center Month Pull-Down List

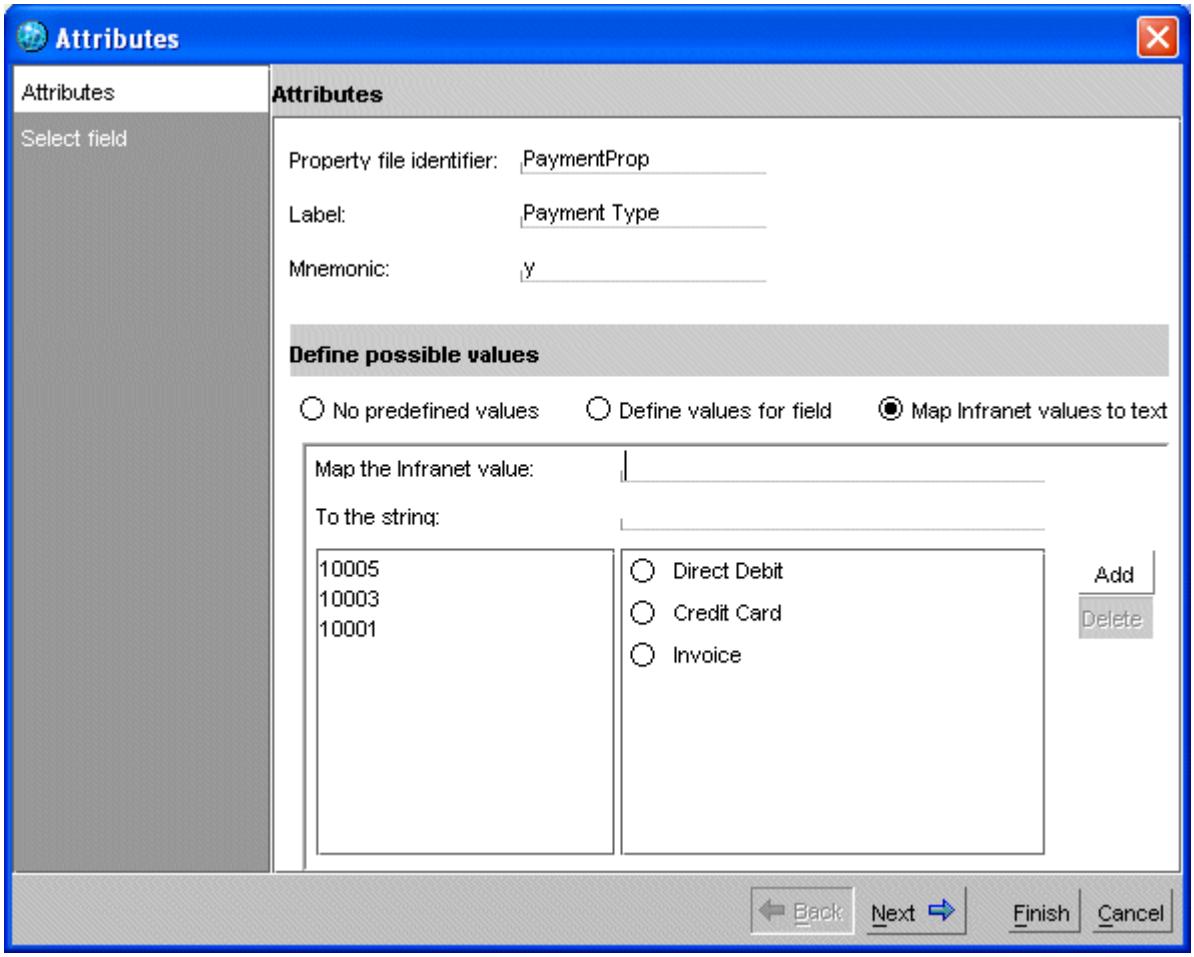


> **Note:**
>
> If no default value is specified and the CSR selects it (the **Not specified** option) the search ignores the field.

4. (Optional) To require CSRs to select from a set of predefined values that map to BRM values:
   a. Click the **Map Infranet values to text** option.
   b. Type value pairs for the **Mapped the Infranet value** and **To the string** fields.
   c. Click **Add**.
   d. Repeat steps 4b and 4c until all options are added.
   e. (Optional) To specify a default value for the field, select the button to the left of the **String** field value.
   f. To delete a field value option, select its row and click **Delete**.

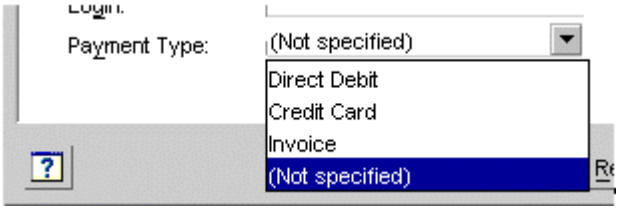
Figure 48-4 shows how to add a **Payment Type** field with selectable values:

Figure 48-4 Adding a Payment Type in Configurator



When a CSR searches on the **Payment Type** field, the pull-down list of payment methods is presented as shown in Figure 48-5:

Figure 48-5 Customer Center Payment Type Pull-Down List



5. Click **Next**.

   The Select field window displays storable classes and their fields in tree format.

6. Select a field from the appropriate storable class.

7. Click **Finish**. The new field appears at the bottom of the list in the Search Criteria section of the main Account Search Configurator window.

Modifying a Search Criteria Field

To modify an existing Search Criteria field:

1. In the Search Criteria area of the Account Search Configurator window, click the field to modify.
2. Click **Modify**.
3. In the Modify wizard, change the displayed attributes of the search field.

> **Note:**
>
> The Modify wizard is similar to the Add wizard. See "Adding a New Search Criteria field". The Modify wizard doesn't include a **Property file identifier** field in the Attributes window. You can't change the **Property file identifier** for a field after you add the field.

Deleting a Custom Search Criteria Field

To delete a custom Search Criteria field:

1. In the Search Criteria area of the Account Search Configurator window, select the custom field to delete.
2. Click **Delete**.

> **Note:**
>
> You cannot delete a default Search Criteria field. However, you can prevent it from displaying in Customer Center by clearing the check box next to the field. You can disable the display of custom Search Criteria in the same way.

Tab Options

This section describes how to reorder, modify, or add a page to the Customer Center account maintenance interface.

1. Start Configurator.
2. Click the **Tab options** button.

   The Configure Page Maintenance wizard appears.
3. Make your configurations. See:
   - Reordering Pages
   - Modifying Attributes of an Existing Page
   - Hiding an Existing Page
   - Adding a New Page
   - Removing a Custom Page

Reordering Pages

To reorder pages, highlight the page's name in the list and click one of the arrows to move its position on the page.

Modifying Attributes of an Existing Page

1. Highlight an existing page in the list.
2. Click **Modify**.
3. Modify the page's name, the tool tip text, or the name of the Java class that creates the page, as required.

Hiding an Existing Page

To hide an existing page, clear its checkbox in the list.

> **Note:**
>
> You cannot delete the default pages included with Customer Center. However, you can prevent their display.

Adding a New Page

To add a new page, click the **Add New** button and enter information listed in Table 48-13:

> **Note:**
>
> To add a new page, first create the new page with an IDE tool such as JBuilder. See "Setting Up JBuilder to Customize the Customer Center Interface".

Table 48-13 New Page Information

Field	Description
Property file identifier	The property file key to attach your class values to.
Name	The tab text to display for your new page's name.
Tool tip	The tool tip text to display when the mouse hovers over the tab name.
Java class name	The name of the source file for your new page, in the format *com.yourpackage.YourClassName*.

Removing a Custom Page

To remove a custom page, highlight the page and click the **Delete** button.

> **Note:**
>
> You cannot delete the default pages included with Customer Center, however, you can prevent their display.

Configuring the Customer Center New Accounts Wizard

To configure fields and panels used in the Customer Center new accounts wizard:

1. Click the **New account** toolbar button in Configurator.

   The new accounts configurator wizard appears.

2. Modify the configurator options as required.

> **Note:**
>
> The selections on each tab in this configuration wizard only affect the fields and options displayed in the **New Account** wizard. They do not impact the **Contact Information** drill-down on the account maintenance **Summary** page.

Table 48-14 describes the main new account configuration pages:

Table 48-14 Contents of the New Account Configuration Page

New Account Wizard Panels	Panel Description
Contacts Panel	Configures options in the Customer Center New account **Contacts** panel
General Panel	Configures options in the Customer Center New account **General** panel
Payment Panel	Configures options in the Customer Center New account **Payments** panel
Billing Panel	Configures options in the Customer Center New account **Billing** panel

To reorder, hide, or add New Account wizard panels, see "New Account Page Options".

Contacts Panel

Select **Contacts** in the navigation bar to open the Contacts Configurator. Use Contacts Configurator to configure the options (shown in Table 48-15) in the **New accounts - Contacts** tab in Customer Center.

Note:

- If you clear a checkbox to hide a field, the fields that follow it are moved up in the display.
- The **Last name**, **Address**, and **Phone** fields cannot be removed.

Table 48-15 Contacts Configurator Panel

Item	New Account - Contacts Panel Action	Default
Company	Select or clear the option to display the **Company** field.	**Displayed** (business accounts only)
Job Title	Select or clear the option to display the **Job title** field.	**Displayed** (business accounts only)
Salutation	Select or clear the option to display the **Salutation** field.	**Displayed**
First Name	Select or clear the option to display the **First name** field.	**Displayed**
Middle Name	Select or clear the option to display the **Middle name** field.	**Displayed**
City	Select or clear the option to display the **City** field.	**Displayed**
State	Select or clear the option to display the **State/Province** field.	**Displayed**
Zip	Select or clear the option to display the **ZIP/Postal** field.	**Displayed**
Country	Select or clear the option to display the **Country** field.	**Displayed**
E-mail	Select or clear the option to display The **E-mail** field.	**Displayed**
Contact Type	Enter the default value for the Customer Center **Contact Type** field. **Note**: The default value appears during account creation and account maintenance operations.	**Account holder**

General Panel

Click the **Next** button or select **General** in the navigation bar to open the General Configurator. Use General Configurator to configure the options (shown in Table 48-16) in the **New accounts - General** tab in Customer Center:

Table 48-16 Items in General Panel for New Account

Item	New Account-General Panel Action	Default
Security Code 1	Select an option to set permissions for the first **Security code** field. • **Editable** • **Not editable** • **Do not display field**	**Editable**
Security Code 2	Select an option to set permissions for the second **Security code** field. • **Editable** • **Not editable** • **Do not display field**	**Editable**
Language (locale) field	Select an option to set permissions for the **Language** field. • **Editable** • **Not editable** • **Do not display field** **Note**: If either the **Read only** or **Hide** options are selected, the user is given a choice at runtime to select the default locale to use.	**Editable**
Currency field	Select an option to set permissions for the **Primary currency** and **Secondary currency**: • **Editable** • **Not editable** • **Do not display field** Select a default entry to use for the **Primary currency** field. Select a default entry to use for the **Secondary currency** field. **Note**: The **Secondary currency** is valid only if the **Primary currency** is **Euro**.	**Editable** **Note**: The default currencies appropriate for the application's locale are displayed.
Hierarchy support	Select or clear the option to display the Hierarchy Setup panel.	**Displayed**
Hierarchy appears initially expanded	Select or clear the option to display the hierarchies in expanded format.	**Displayed**
Sponsorship support	Select or clear the option to display the Sponsorship Setup panel.	**Displayed**
Sponsorship appears initially expanded	Select or clear the option to display the sponsorship in expanded format.	**Displayed**

Payment Panel

Click the **Next** button or select **Payment** in the navigation bar to open the Payment Configurator. Use Payment Configurator to configure the options (shown in Table 48-17) in the **New accounts - Configurator** tab in Customer Center:

Table 48-17 Payment Panel Settings

Item	New Account Wizard - Payment Panel Action	Default
Default payment type - consumer	Select a default payment method from the drop-down menu.	**Invoice**
Default payment type - business	Select a default payment method from the drop-down menu.	**Invoice**

Billing Panel

Click the **Next** button or select **Billing** in the navigation bar to open the Billing Configurator. Use Billing Configurator to configure the options (shown in Table 48-18) in the **New accounts - Billing** tab in Customer Center:

Table 48-18 Billing Panel

Item	New Account Wizard – Billing Panel Action	Default
Accounting type field	Select one of these options to specify access to accounting type: • **Editable** • **Not Editable** • **Do not display field**	**Editable**
Accounting type	Select one of these options to specify access to accounting type: • **Balance forward** • **Open Item**	**Balance forward**

New Account Page Options

This section describes how to reorder, modify, or add a page, or add a profile panel to the Customer Center New Account wizard interface.

1. Start Configurator.
2. From any New account panel, select the **Page Options** button at the bottom of the panel.
3. Select either **Consumer account** or **Business account** depending on the account type you want to configure.
4. Make your configurations. See:
   - Reordering New Account Pages
   - Modifying an Existing Page
   - Hiding an Existing Page
   - Adding a New Page
   - Removing a Custom Page

Reordering New Account Pages

To reorder pages, highlight the page's name in the list and click one of the arrows to change its position on the page.

Modifying an Existing Page

1. Highlight an existing page in the list.
2. Click **Modify**.
3. Modify the page's name, or the Java class that creates the page, as required.

Hiding an Existing Page

To hide an existing page, clear its checkbox in the list.

> **Note:**
>
> You cannot delete the default pages included with Customer Center. However, you can prevent their display.

Adding a New Page

To add a new page, click the **Add New** button and enter information in the fields shown in Table 48-19:

> **Note:**
>
> To add a new page, first create the new page with an IDE tool such as JBuilder. See "Setting Up JBuilder to Customize the Customer Center Interface".

Table 48-19 Information Required for a New Page

Field	Description
Property file identifier	The property file key to attach your class values to.
Name	The tab text to display for your new page's name.
Java class name	The name of the source file for your new page, in the format *com.yourpackage.YourClassName*.

Removing a Custom Page

To remove a custom page, highlight the page and click the **Delete** button.

> **Note:**
>
> You cannot delete the default pages included with Customer Center, however, you can prevent their display.

Using the Configurator Resource String Editor

Configurator includes a Resource String Editor. You use this feature to replace Customer Center field labels with your custom text values.

To use the Resource String Editor, follow these steps:

1. Starting the Resource String Editor
2. Searching for Labels to Replace
3. Replacing Labels with New Strings
4. Undoing Label Changes

Starting the Resource String Editor

To start the Resource String Editor, choose **Tools - Resource String Editor** from the Configurator main menu. The Resource String Editor appears.

Searching for Labels to Replace

To locate Customer Center field labels whose text you want to replace, type the label text you want to locate in the **Change text from** field and click **Search**. Any labels that contain matching text will appear in the results area.

Resource String Editor String Search Rules

- Search matching is case sensitive.
- Search matching is performed against the whole value of the field. For example, if you search for the string Balance, the results include screen labels that consist of only the word Balance.
- You can use the use the wildcard symbol (*****) at the end or beginning of your search string to return all labels that *start* or *end* with the search string, respectively.
  - **Example 1**

    If you search for the string **Balance***, you get the same matches as you do if you search for **Balance** plus longer labels that *start* with **Balance**, such as **Balance summary** and **Balance forward**.
  - **Example 2**

    If you search for ***forward**, two items called **Balance forward** are returned.

Replacing Labels with New Strings

To replace label text with new text:

1. Click one or more matching fields whose label text you want to replace.
2. In the **Change text to** field, type the replacement text.

> **Note:**
>
> The *entire label string* of the selected labels is replaced with the replacement text in the **Change text to field**, not just the string you searched for.

3. Click **Apply**.

Undoing Label Changes

To undo the last string replace operation during the current Configurator session:

1. If it is not already running, start Resource String Editor.
2. Click **Undo**.
3. When prompted to confirm, click **Yes**. Your changes are reversed.

Additional Configured Profile Panel Examples

For more complete examples, see the code in *CCSDK_home*/**CustomerCareSDK/CustCntrExamples/Profile:**

- **CSProf.java** demonstrates how to extend this example to manipulate the list of plans available for sale.
- **CDProf.java** is an example similar to the one discussed above, but uses a different **/profile** object and different BAS widgets.

49

Adding Custom Fields to Customer Center

Learn how to add custom fields to your Oracle Communications Billing and Revenue Management (BRM) Customer Center implementation.

Topics in this document:

- Coding and Deploying Custom Fields for Customer Center
- Adding Custom Fields to Infranet.properties
- Generating Your Custom Field Java Source Code
- Compiling and Signing Your Custom Fields Java Source Code
- Configuring JBuilder to Add Custom Fields to Customer Center
- Building and Deploying Your New Profile Panel

You should have a working knowledge of JBuilder.

Coding and Deploying Custom Fields for Customer Center

To code and deploy custom fields for Customer Center, perform these tasks in this order:

1. Adding Custom Fields to Infranet.properties
2. Generating Your Custom Field Java Source Code
3. Compiling and Signing Your Custom Fields Java Source Code
4. Building and Deploying Your New Profile Panel

Adding Custom Fields to Infranet.properties

As with all BRM Java clients, the field numbers for Customer Center custom fields must be added to **Infranet.properties**. If you add custom fields to Customer Center:

1. Add your custom field numbers to the **Infranet.properties** file.
2. Before creating your **jar** file, copy the modified **Infranet.properties** file to the top level of the directory structure along with your other compiled source files.

> **Note:**
>
> When Customer Center is deployed by using Web Start, an **Infranet.properties** file is normally not required since the BRM host and port information is read from the Web Start **.jnlp** file. However, if the **Infranet.properties** file exists in the CLASSPATH, it is recognized by Customer Center, or, more specifically, Portal Communication Module (PCM).

For example, if you are adding a custom credit score field and a custom panel for displaying that field, the file and directory structure might look like this:

- **./Infranet.properties**
- **./com/**
- **./com/mycompany/**
- **./com/mycompany/CustomPanel.class**
- **./customfields/**
- **./customfields/CreditScore.class**

Note that the **Infranet.properties** file is at the top level of this directory structure.

This **jar** command:

```
jar cvf ccCustomFields.jar .
```

Generates a **jar** file containing:

- 0 Mon Dec 10 09:03:40 PST 2002 META-INF/MANIFEST.MF
- 0 Mon Dec 10 08:58:28 PST 2002 **Infranet.properties**
- 0 Mon Dec 10 09:01:14 PST 2002 **com/**
- 0 Mon Dec 10 08:58:54 PST 2002 **com/mycompany/**
- 0 Mon Dec 10 08:58:54 PST 2002 **com/mycompany/CustomPanel.class**
- 0 Mon Dec 10 09:01:36 PST 2002 **customfields/**
- 0 Mon Dec 10 09:01:36 PST 2002 **customfields/CreditScore.class**

> **Note:**
>
> – Be sure to include the "." at the end of the **jar** command. This specifies the current directory.
>
> – This example assumes that you have only one **Infranet.properties** file packaged in a **jar** file per Customer Center deployment. The **jar** file is recognized and picked up when Customer Center is run.

For detailed custom field development procedures, see "Adding Custom Fields to Customer Center".

Generating Your Custom Field Java Source Code

This section describes how to select the custom fields profile object that you want to add to Customer Center.

> **Note:**
>
> This procedure uses the sample profile object **/profile/customfieldsprof**. Replace this object name with the one you create in Developer Center.

1. Start Storable Class Editor in Developer Center.
2. Create your custom fields in a profile object.

For information, see the Storable Class Editor Help.

3. Create a directory in the **CustCntr/custom** directory for your custom field Java code, as in this example:

```
mkdir CustCntr\custom\custom_fields
```

4. Choose **File – Generate Custom Fields Source**.
5. Click **Browse** and select the **CustCntr/custom/custom_fields** directory.

> **Note:**
>
> If you are deploying only in Java, you can deselect the option to export C code.

6. Click **OK**.

   The source Java code for your custom fields is generated in **CustCntr/custom/custom_fields**. In this example, these files are:

   - **InfranetPropertiesAdditions.properties**
   - **XCreditScore.java**
   - **XLicenseNum.java**

7. Click **OK** in the confirmation dialog box.

Compiling and Signing Your Custom Fields Java Source Code

This section describes how to compile and sign the profile object that you created by using the Storable Class Editor.

1. Go to the **CustCntr/custom/custom_fields** directory.
2. Copy the contents of the custom fields file (*CCSDK_home*/**CustomerCareSDK/CustCntr/custom/custom_fields**) to the end of the *CCSDK_home*/**CustomerCareSDK/CustCntr/bin/Infranet.properties** file by using a text editor.

> **Note:**
>
> The **InfranetPropertiesAdditions.properties** file contains information required when running code that references your custom fields.

3. Rename the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/custom_fields/InfranetPropertiesAdditions.properties** file to **Infranet.properties**.

   You use this file for deploying your custom fields with WebStart. This file is similar to the one you created in step 2 except it doesn't contain the connection and login parameters.

4. If you have more than one set of custom fields, merge all associated **InfranetPropertiesAdditions.properties** files.

> **Note:**
>
> If you are deploying multiple sets of custom fields (you used Developer Center to generate multiple custom source files), you have several copies of the **InfranetPropertiesAdditions.properties** file. You must merge all of these files into the *CCSDK_home*/**CustomerCareSDK/CustCntr/bin/Infranet.properties** file.

5. Compile your custom fields.
   a. Go to your custom field directory (*CCSDK_home*/**CustomerCareSDK/CustCntr/custom/custom_fields**).
   b. Compile your source code:

      ```
      javac -classpath ../../../lib/pcm.jar -d . *.java
      ```

      A directory matching the Java package name of your custom fields (in this example, **customfields**) is created.
6. Package the compiled code and the revised **Infranet.properties** file, as in this example:

   ```
   jar cf ../ccCustomFields.jar customfields Infranet.properties
   ```

   The **jar** file is placed up one level in the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory.
7. If you are deploying the **jar** file using Web Start, sign the **jar** file.
   a. Go to *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory and find your **jar** file.
   b. Sign your **jar** file by using the **signjar** script, as in this example:

      ```
      signjar.bat ccCustomFields.jar
      ```

   c. Verify that the signing process is completed properly by using the **jarsigner** utility provided in the JDK:

      ```
      jarsigner -verify ccCustomFields.jar
      ```

      The expected output is:

      ```
      jar verified.
      ```

What's Next

See "Configuring JBuilder to Add Custom Fields to Customer Center".

Configuring JBuilder to Add Custom Fields to Customer Center

This section describes how to configure your custom fields in JBuilder. This enables you to use the fields later when building a new panel or other customization.

Before you use JBuilder to configure your custom fields, set up a JBuilder project. See "Setting Up JBuilder to Customize the Customer Center Interface".

1. Start JBuilder.
2. Open your BRM project. See "Creating a JBuilder Project for Customer Center SDK".

3. Chose **Project – Project Properties**.
4. Click the **Required Libraries** tab.

   You should see a single library named CCSDK.
5. Select the **CCSDK** library and then click **Edit**.
6. In the Configure Libraries dialog box, click **Add**.
7. Go to your CCSDK install directory, select your custom fields **jar** file, and then click **OK**.
8. Click **OK** in each of the open dialog boxes.
9. Go to the *CCSDK_home*/**CustomerCareSDK/CustCntrExamples/Profile** directory.
10. Copy **ProfileTemplate.txt** to your custom fields file (**CustomFieldsProfile.java** in this example).
11. Choose **File - Open File** and open the **CustomFieldsProfile.java** file.
12. Replace the two instances of XXX with your Java Class name (**CustomFieldsProfile** in this case).
13. In **CustomFieldsProfile.java**, find this line:

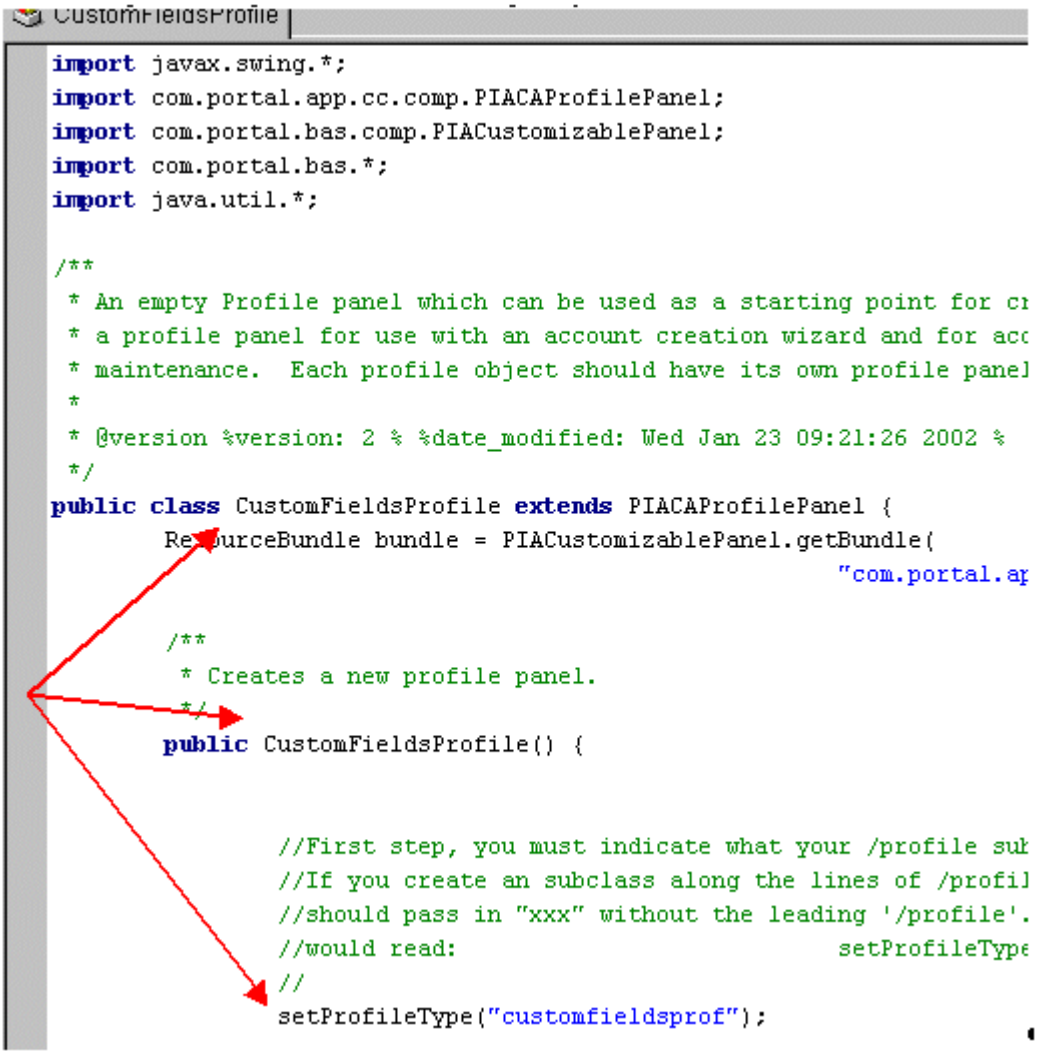
    ```
    setProfileType();
    ```

    And specify the **customfieldsprof** subclass:

    ```
    setProfileType(customfieldsprof);
    ```

    This maps to the profile object created in Developer Center. This example shown in Figure 49-1 references the **/profile/customfieldsprof** profile object.

Figure 49-1 CustomFieldsProfile Example



14. Click the **Design** tab at the bottom of the JBuilder window.
15. Verify that you can launch the Business Application SDK (BAS) widget Configurator and that it displays your custom profile object and custom fields.
    a. In the widget palette at the top of the JBuilder window, click the **CCSDK Components** tab.
    b. In the lower left pane, click the PIATextfield widget. This is typically the first widget in the list.
    c. Drag the widget into your panel.
    d. Right-click the widget instance on the left side of the window and click Configurator.
    e. If prompted, log in to BRM.
    f. Expand the node containing your custom field and select your profile object.

    If your profile object is visible in the configurator, you have successfully made JBuilder aware of your custom fields. You can continue constructing your panel. If it is not visible verify that:

    - Your JBuilder library path is correct.

- There are no errors in the **Infranet.properties** file you modified in **CustCntr/bin** when you combined the existing file with the **InfranetPropertiesAdditions.properties** file that was generated by Developer Center.
- Your **CustCntr/bin/Infranet.properties** file contains the entries for your custom fields. Choose **Project – Project Properties** in JBuilder, click the **Required libraries** tab select **CCSDKlib** in the list, and click **Edit**. Verify that **CustCntr/bin** and **ccCustomFields.jar** are in the list.

16. When you have completed your panel layout, save your changes.

What's Next

See "Building and Deploying Your New Profile Panel".

Building and Deploying Your New Profile Panel

This section describes how to build and deploy your custom fields in your custom profile panel:

1. Run Configurator to add your profile panel to Customer Center, and save your changes:
   - If you are adding your new panel to a Customer Center account maintenance page, see "Tab Options".
   - If you are adding your new panel to a Customer Center new accounts wizard page, see "New Account Page Options".
2. Run the build script (*CCSDK_home/***CustomerCareSDK/buildAll.bat**). This script compiles and packages your new panel into the *CCSDK_home***/CustomerCareSDK/CustCntr/custom/ccCustom.jar** file.
3. Add **ccCustomFields.jar** to the CLASSPATH in the *CCSDK_home***/CustomerCareSDK/CustCntr/bin/runCustomerCenter**:

   ```
   set CLASSPATH=f:/7.2sdk/CustCntr/custom/ccCustomFields.jar;%CLASSPATH%
   ```

4. Test your customizations. See "Testing Your Customizations".
5. To deploy your customizations with Customer Center WebStart:
   a. Open the *CCSDK_home***/CustomerCareSDK/CustCntr/custom/custom.jnlp** file with a text editor.
   b. Add this entry for **ccCustomFields.jar:**

      ```
      <jar href="custom/ccCustomFields.jar"/>
      ```

   c. Continue with the standard deployment procedure. See "Deploying Your Customer Center Customizations".

50

Setting Up JBuilder to Customize the Customer Center Interface

Learn how to set up JBuilder to customize Oracle Communications Billing and Revenue Management (BRM) Customer Center functionality.

Topics in this document:

- About Using JBuilder to Customize the Customer Center Interface
- Adding PIA Widgets to the JBuilder Palette
- Creating a JBuilder Project for Customer Center SDK

About Using JBuilder to Customize the Customer Center Interface

To set up your JBuilder development environment for customizing Customer Center:

1. Make sure you have installed the following on your system:
   - JDK
   - JBuilder

   For more compatibility information, see *BRM Compatibility Matrix*.
2. Install Customer Center SDK. See *BRM Installation Guide*.
3. Add PIA widgets to the JBuilder palette. See "Adding PIA Widgets to the JBuilder Palette".
4. Create a JBuilder project. See "Creating a JBuilder Project for Customer Center SDK".

Adding PIA Widgets to the JBuilder Palette

Customer Center SDK includes a JBuilder plug-in of Privacy Impact Assessment (PIA) widgets. You can use these widgets to add components to existing Customer Center panels or to custom panels you create by subclassing the controllers provided with the SDK.

To add PIA widgets to the JBuilder palette, copy the Customer Center SDK plug-in file (*CCSDK_home*/**CustomerCenterSDK/CustCntr/bin/oracle.communications.brm.uicomponents_1.0.0.jar**) to your JBuilder directory.

Creating a JBuilder Project for Customer Center SDK

To create a JBuilder project for Customer Center SDK:

1. Start JBuilder.
2. Choose **File – New – Java Project**.

   The New Java Project wizard starts.

3. In the **Project Name** field, enter a name for your project.
4. Select the appropriate JRE.
5. Set the remaining fields according to your system configuration.
6. Click **Next**.

   The Java Settings pane appears.
7. Click the **Libraries** tab and then click **Add Library**.

   The Add Library dialog box appears.
8. In the Add Library pane, select **User Library** and then click **Next**.

   The User Library pane appears.
9. Click **User Libraries**.

   The Preferences (Filtered) dialog box appears.
10. In the User Libraries pane, click **Import**.

    The Import User Libraries dialog box appears.
11. In the **File location** field, enter the path to the Customer Center SDK library file (*CCSDK_home*/**CustomerCenterSDK/CustCntr/bin/CCSDK_lib.userlibraries**).
12. Click **OK**.

    The User Library dialog box appears.
13. Make sure **CCSDK_lib** appears in the **User libraries** list.

> **Note:**
>
> If **CCSDK_lib** is not in the **User libraries** list, the **CCSDK_lib.userlibraries** file is not in a folder that JBuilder can access. See "Adding PIA Widgets to the JBuilder Palette" for information on where to copy the file.

14. Select **CCSDK_lib** and then click **Finish**.

    Your project is now ready for adding Java source files.

51

Creating a New Customer Center Service Panel

Learn how to create a new Oracle Communications Billing and Revenue Management (BRM) Customer Center service panel using JBuilder.

Topics in this document:

- Creating a New Service Panel

> **Note:**
>
> These instructions assume you have already set up JBuilder and have a working knowledge of JBuilder.

Creating a New Service Panel

Service panels are extensions to the **PIAExtendServiceBase** class. They are used to add data entry or display fields to the account maintenance service tab. The fields displayed are dependent on the type of service selected.

> **Note:**
>
> You mostly work with the widgets in the CCSDK Components widget palette since they are BRM aware. Additional widgets are included under the PFC **Components** tab, but they are not BRM aware. They provide functionality beyond the normal Java widget set, such as links, section headers, and drop-down menus.

As an example, these instructions create a panel for the email service. This panel includes three fields:

- **Maximum message size**
- **Mailbox path**
- **Service status**

> **Note:**
>
> Customer Center displays status information elsewhere in the UI. The service status field is added here as an example only.

Follow these steps to create a new service panel:

1. Go to the *CCSDK_home*/**CustomerCenterSDK/CustCntrExamples/Service** directory.
2. Copy **ServiceTemplate.txt** to **CustomService.java**. You can give the file any name.
3. Edit **CustomProfile.java** by changing all instances of XXX to **CustomService** (or whatever you have named your file) and move this file to the *CCSDK_home*/**CustomerCenterSDK/CustCntr/custom** directory.

> **Note:**
>
> Always develop your deployment code in the *CCSDK_home*/**CustomerCenterSDK/CustCntr/custom** directory.

4. Start JBuilder and load the project you previously created.

   The example project is called MyCustomizations.
5. Choose **File – Open** to open the **CustomProfile.java** file.
6. (Optional) Include code to inform the base class that you are handling the login and password data entry elsewhere.

   You might want full control over how the login and password data are entered. For example, some services store non-traditional information in the **login** and **password** fields, such as phone numbers and URLs. In these situations, you should change the UI field labels accordingly. If you fill in these fields automatically, you may also not want to display these fields at all.

   To instruct the base class that you are handling login and password data entry elsewhere:

   a. In JBuilder, click the **Source** tab.
   b. Add the following code towards the bottom of your source file:

   ```
   public boolean supportsLoginAndPassword() {
   return true;
   }
   ```

   c. Save the file.

> **Note:**
>
> - Your UI must accommodate these fields with fields you add to the UI and label yourself or by passing the appropriate data directly to the input flist.
> - This example assumes that login and password entry is done in the traditional sense. In this situation, you do not need to add these fields to your UI.

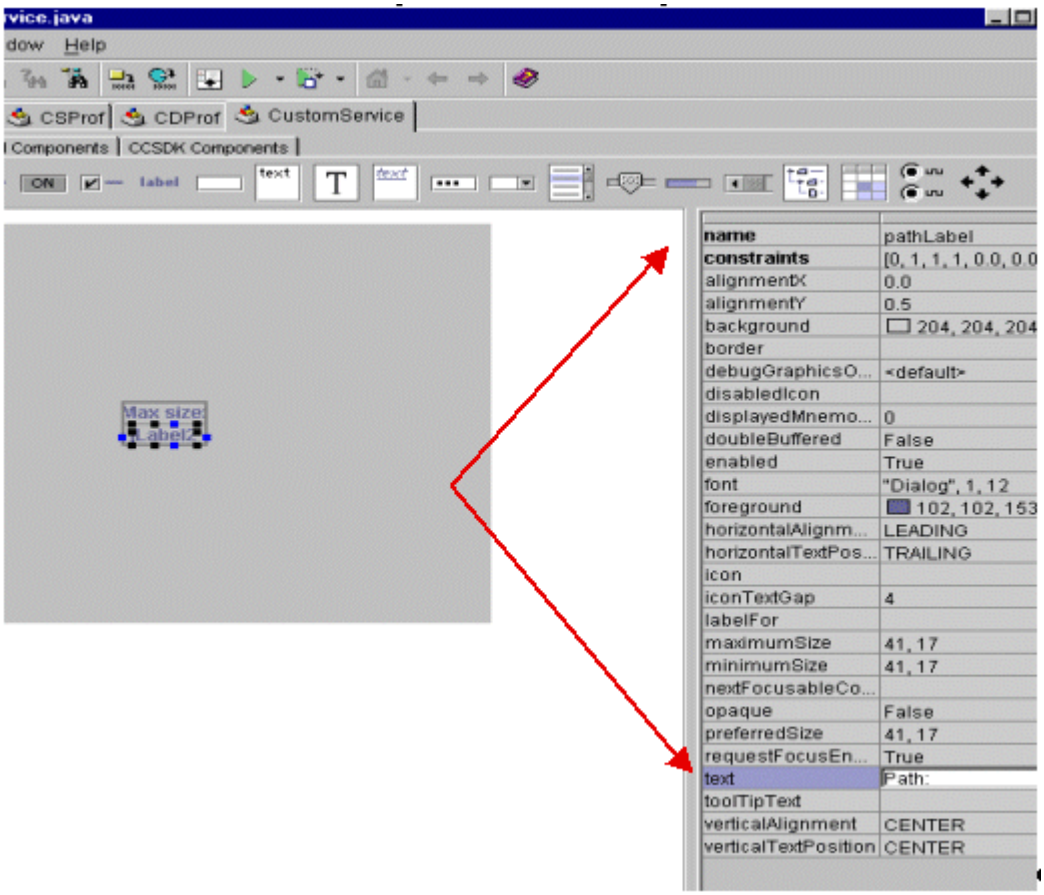
7. Click the **Design** tab.
8. Select the gray panel in the center of the screen and make sure the layout is set to GridBagLayout.

   If not, choose GridBagLayout from the drop-down menu associated with the layout property.
9. From the Swing widget palette, select the Label widget.
10. Drag one label in the gray work area in the center of the screen.

11. Select the label widget again and drag it below the first one.
12. Select the first label, click the name attribute and change it to maxSizeLabel.
13. Click the text attribute and change it to Max size:.
14. Select the second label, click the name attribute and change it to pathLavel.
15. Click the text attribute and change it to Path: as shown in Figure 51-1.

    This represents the mailbox path value.

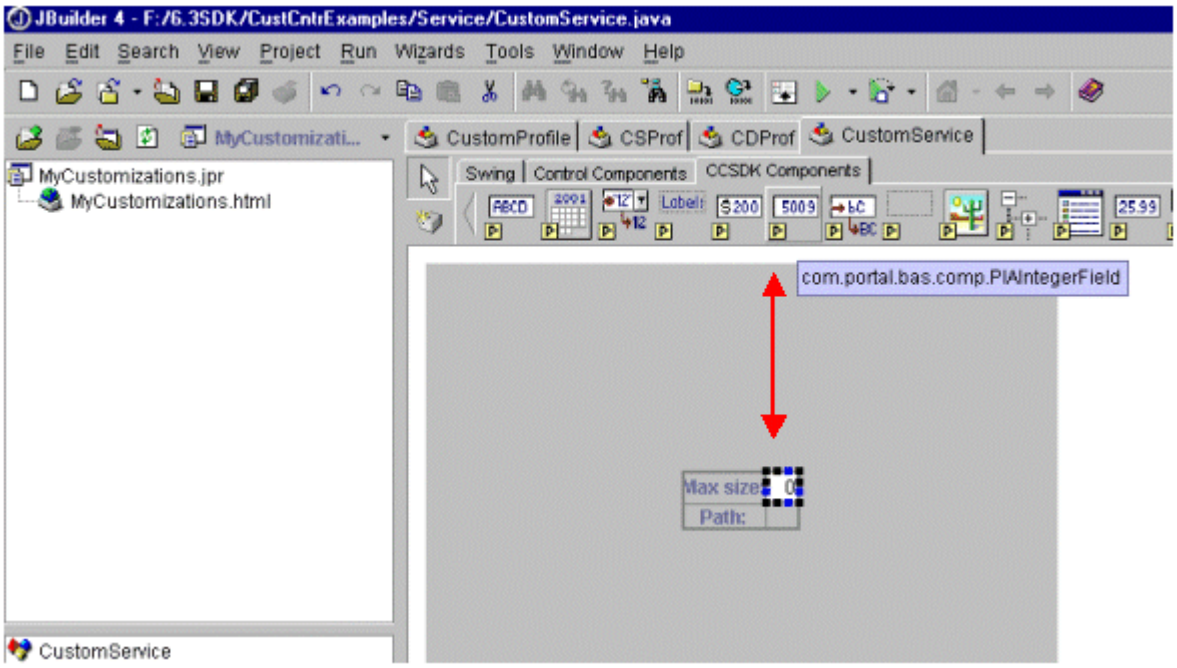
Figure 51-1 Changing the Text Attribute



16. From the CCSDK Components widget palette, select the PIAIntegerTextField widget and drag it to the right of the Max size label widget as shown in Figure 51-2.

    This configures the widget to capture the max size data in an integer-only text entry widget.

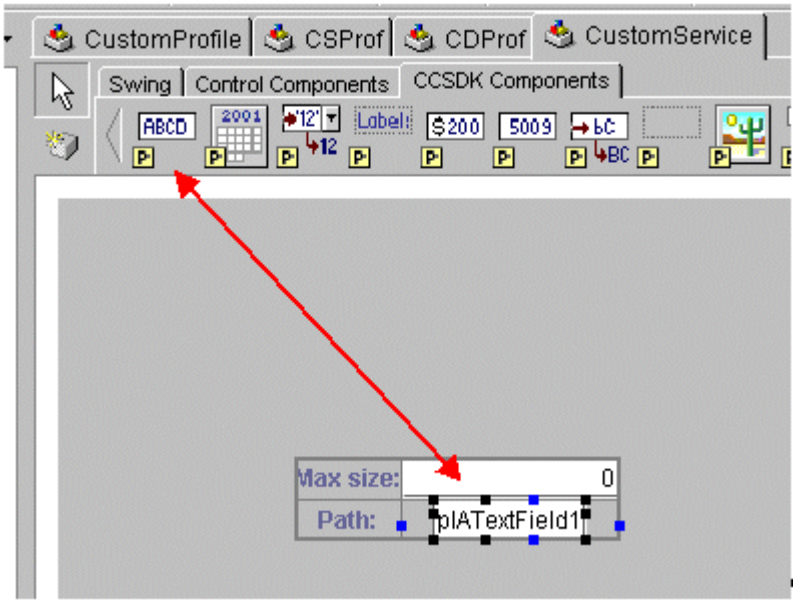
Figure 51-2 Adding a PIAIntegerTextField Widget



17. Select the PIAIntegerField widget and select the columns attribute.
18. Change the columns attribute to **10**.

    This causes the widget width to expand.
19. Right-click the widget and from the menu and set its fill to Horizontal.
20. Select the PIATextField widget and drag it to the right of the Path widget as shown in Figure 51-3.

Figure 51-3 Adding a PIATextField Widget



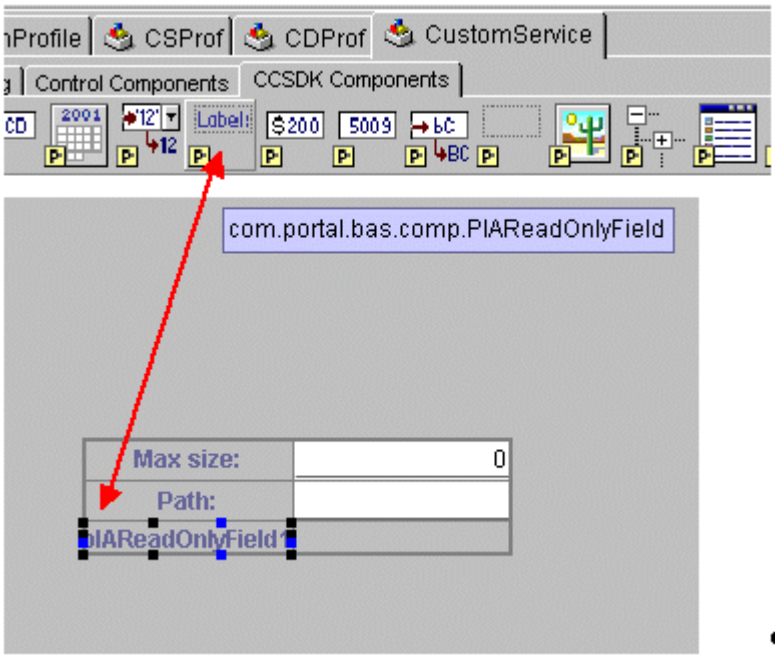
21. Change the widget columns attribute to **10** and set its fill to Horizontal.

22. Select the existing text and delete it.

    This clears the text attribute so it no longer reads pIATextField.

23. Click the PIAReadOnly widget from the CCSDK Components palette and drag it below the Path: label as shown in Figure 51-4.

Figure 51-4 Adding a PIATextReadOnly Widget



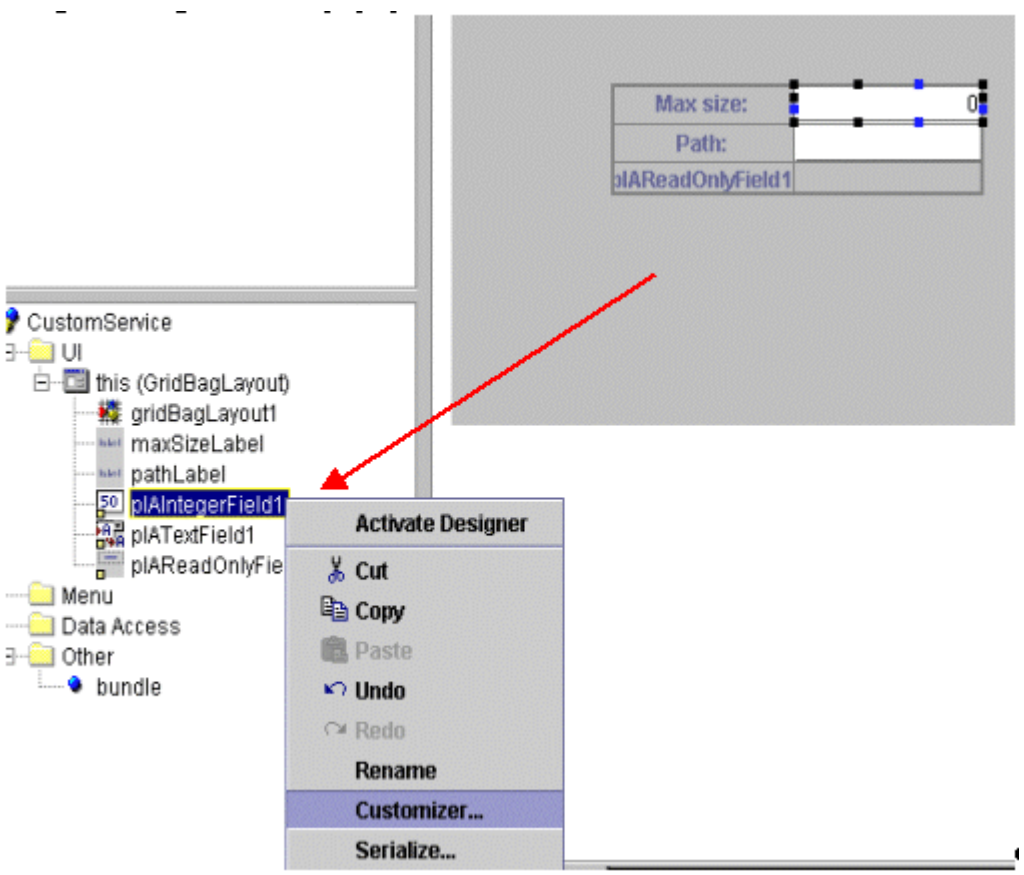
24. Map the widgets to BRM fields:

    a. Click the PIAIntegerTextField representing "Max size" and bring up the Infranet-aware customizer.

    b. Select the variable instance for this widget and right click to popup the action menu.

    c. Select Customizer as shown in Figure 51-5.

Figure 51-5 Selecting Customizer



> **Note:**
>
> Before the customizer appears, you must log in to BRM. Log in as you normally would.

d. From within the customizer, locate the **/service/email** node in the tree and Click the PIN_FLD_MAX_MSG_SIZE field on the right.

e. Select **Apply**, and then **OK**.

   You have now mapped that widget to a field in BRM.

f. Follow the same procedure to map the Path textfield widget to PIN_FLD_PATH.

25. Map the read-only widget to a BRM field:

    a. Launch the read-only widget customizer.

    b. Select that class and select PIN_FLD_STATUS.

> **Tip:**
>
> Since the widget is read-only and its value cannot change, you do not need to work with the ModelFieldDescription. (ModelFieldDescription is what the Business Application SDK (BAS) turns into the input flist when it saves data). You can delete this value.

> **Note:**
>
> The field is not in the **/service/email** class but instead in the base **/service** storable class.

26. Click **Display Field Format**.

    The Display Field Format editor appears.

    > **Tip:**
    >
    > You change the display field format feature to make BRM values easier for the user to read.

27. Enter the BRM data value and the string you want displayed in the UI.
28. When you are finished, click **Done**.

    This updates the BAS widget property with the correct mapping format.
29. (Optional) Add the leading text **Status -** to the front of the display field format.
30. Save your file.

    Your basic service panel is now complete.
31. Compile your panel using the **buildAll** script in the top-level SDK directory.

    See "Building Your Customer Center Customizations".
32. Add this entry to the *CC_SDK*/**CustomerCareSDK/CustCntr/custom/Customized.properties** file:

    ```
    extended.service.email=CustomService
    ```

    Replace *CustomService* with the name of your Java Class.

    This entry makes Customer Center aware of the new service panel.
33. From the *CC_SDK*/**CustomerCareSDK/CustCntr/bin** directory, start your local copy of Customer Center (run **CustomerCenter.bat**).
34. Locate an account containing an email service.
35. Switch to the **Services** tab and select the email service in the table.

    Your panel and the default widget set appear.

If the fields on your service panel need alignment, see "Correcting Field Alignment".

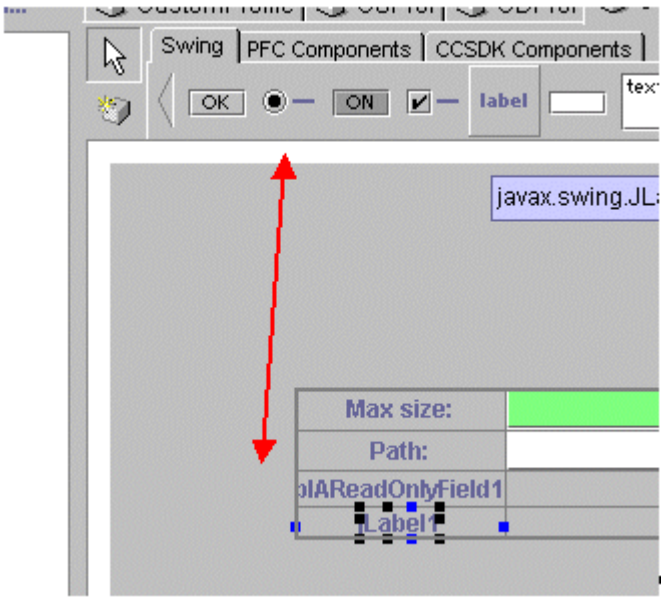
Correcting Field Alignment

If your panel has alignment problems, you might need to work with the Java layout to correct them.

This section describes how to add a blank widget in your panel to fix alignment problems. The widget is invisible but takes up the extra space within the panel. (By default, widgets in a GridBagLayout tend to move towards the center.)

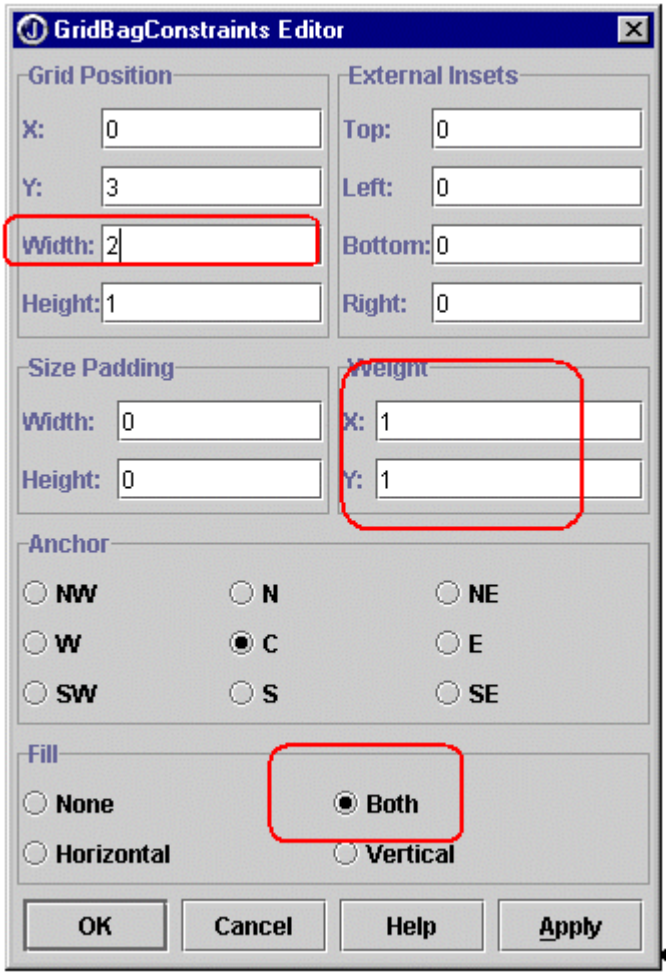
1. Exit Customer Center.
2. Open JBuilder and click the **Design** tab.
3. From the **Swing** widget palette, select the 'label' widget and drop it below your status read-only widget as seen in Figure 51-6.

Figure 51-6 Adding a Label Widget



4. Click the constraints attribute.
5. Select the **...** button to launch the editor.
6. Set the **Grid Position Width** to **2**; set the **Weight X** and **Y** parameters to **1**; set the **Fill** parameter to **BOTH** as seen in Figure 51-7.

Figure 51-7 GridBagConstraints Editor



The first 3 rows of widgets should move toward the top of the panel.

You have just told the label in the fourth row to take up two columns of space, and all remaining space on the bottom.

7. Delete the value in this widget's 'text' attribute so it no longer reads 'label'. Select the 'text' attribute on the right side of JBuilder and delete the text.
8. Save your changes, compile, and re-launch Customer Center. When you revisit your service panel the layout should be improved.

What's Next

If you have no further Customer Center customizations to code, see "Building Your Customer Center Customizations".

52

Creating a New Customer Center Profile Panel

Learn how to create a new Oracle Communications Billing and Revenue Management (BRM) Customer Center profile panel using JBuilder.

Topics in this document:

- Creating a New Profile Panel

> **Note:**
>
> These instructions assume you have already set up JBuilder and have a working knowledge of JBuilder.

Creating a New Profile Panel

Profile panels are extensions to the **PIACAProfilePanel** class. You use them to add data entry or display fields to the Customer Center Account Maintenance and New Accounts pages.

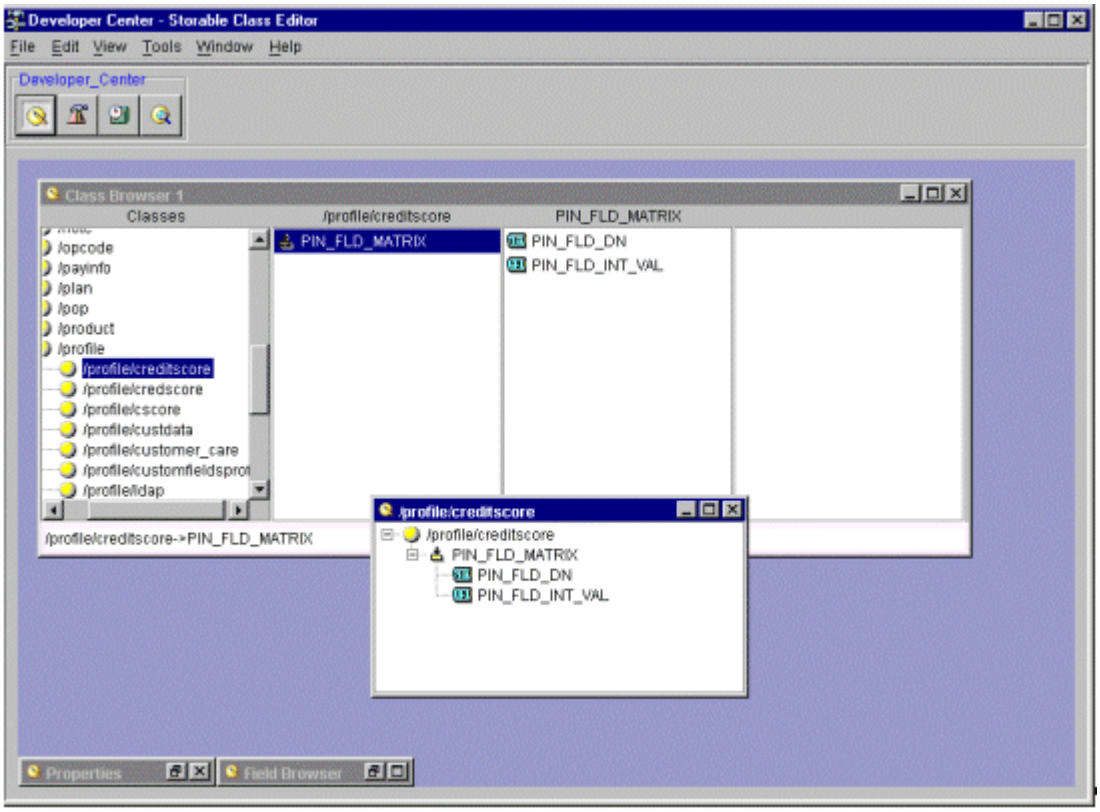
> **Note:**
>
> You mostly work with the widgets in the CCSDK Components widget palette since they are BRM aware. Additional widgets are included under the PFC **Components** tab, but they are not BRM aware. They provide functionality beyond the normal Java widget set, such as links, section headers, and drop-down menus.

To create a new profile panel:

1. Be sure the **/profile/creditscore** profile object has been created in BRM.

   Figure 52-1 shows a sample window from Developer Center. The **CreditScoreProfile.sce** file is open.

Figure 52-1 Developer Center View of /profile/creditscore Object



2. Go to the *CCSDK_home*/**CustomerCenterSDK/CustCntrExamples/Profile** directory.
3. Copy **ProfileTemplate.txt** to **CustomProfile.java**. You can give the file any name.
4. Edit **CustomProfile.java** by changing all instances of XXX to **CustomProfile** (or whatever you have named your file) and move this file to the *CCSDK_home*/**CustomerCenterSDK/CustCntr/custom** directory.

> **Note:**
>
> Always develop your deployment code in the *CCSDK_home*/**CustomerCenterSDK/CustCntr/custom** directory.

5. Find the following line of code:

```
setProfileType();
```

and modify this line to call out your profile subclass. For this example, change the line to:

```
setProfileType("creditscore");
```

> **Note:**
>
> Be sure to omit the **/profile**.

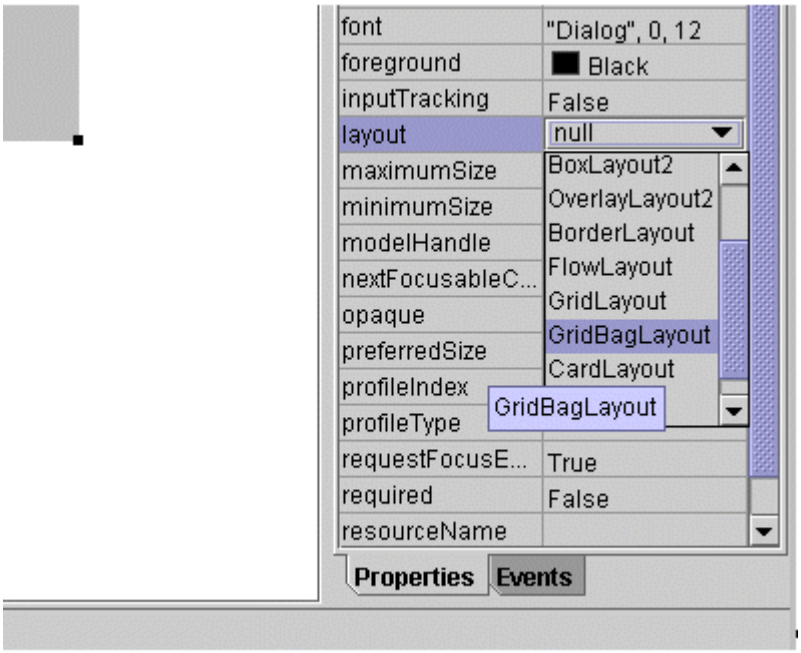
6. Start JBuilder and load the project you previously created.

The example project is called MyCustomizations.

7. Choose **File – Open** to open the **CustomProfile.java** file, and then click the **Design** tab.
8. Select the gray panel in the center of the screen and make sure the layout is set to GridBagLayout.

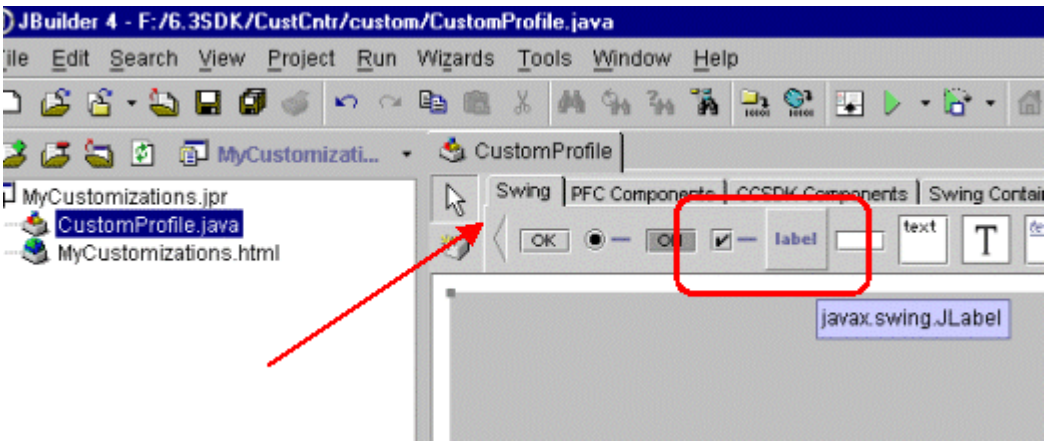
   If not, choose GridBagLayout from the drop-down menu associated with the layout property as shown in Figure 52-2.

Figure 52-2 Setting GridBagLayout



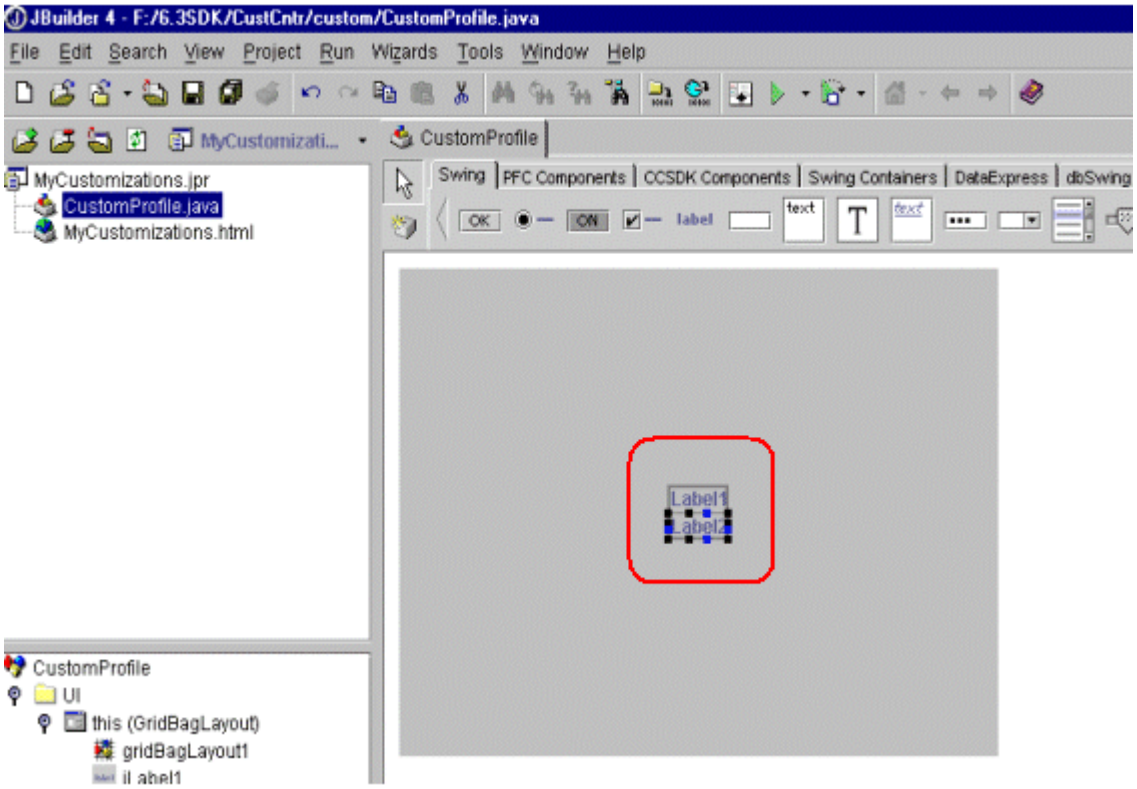
9. From the Swing widget palette, select the Label widget as shown in Figure 52-3.

Figure 52-3 Selecting the Swing Label Widget



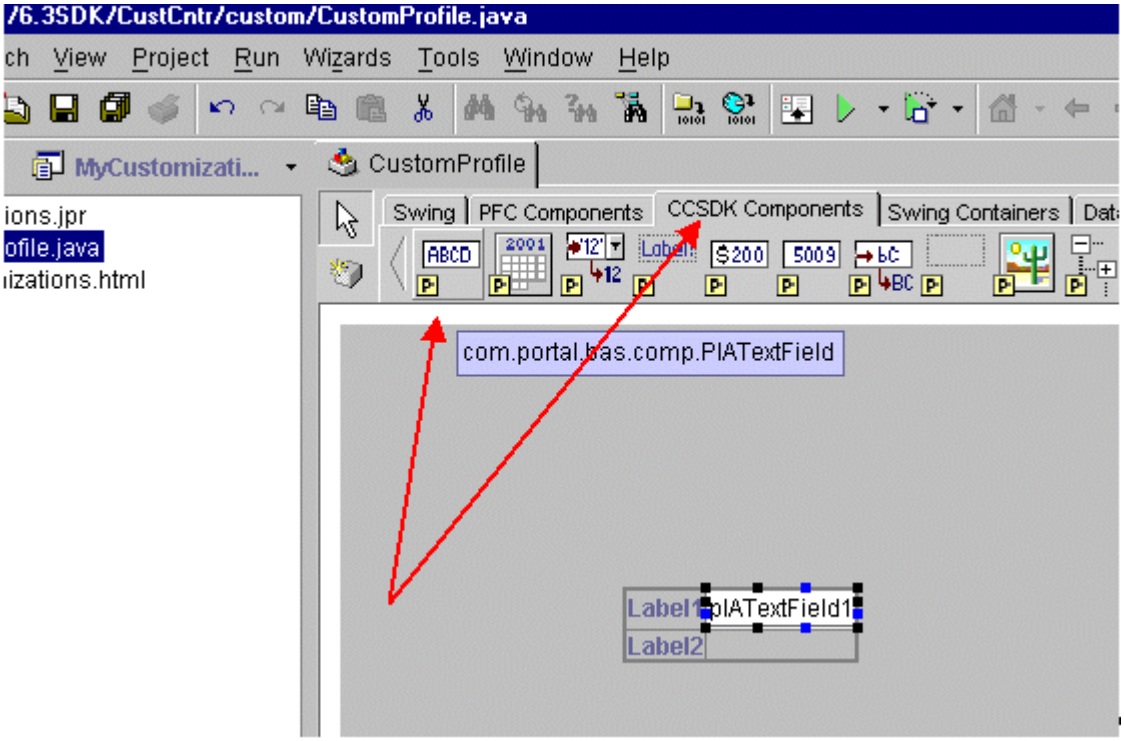
10. Drag a label widget in the gray work area in the center of the screen.
11. Select the label widget again and drag it below the first one as shown in Figure 52-4.

Figure 52-4 Adding a Second Label Widget



12. From the widget palette on the **CCSDK Components** tab, drag the PIATextField widget next to the first label widget as shown in Figure 52-5.

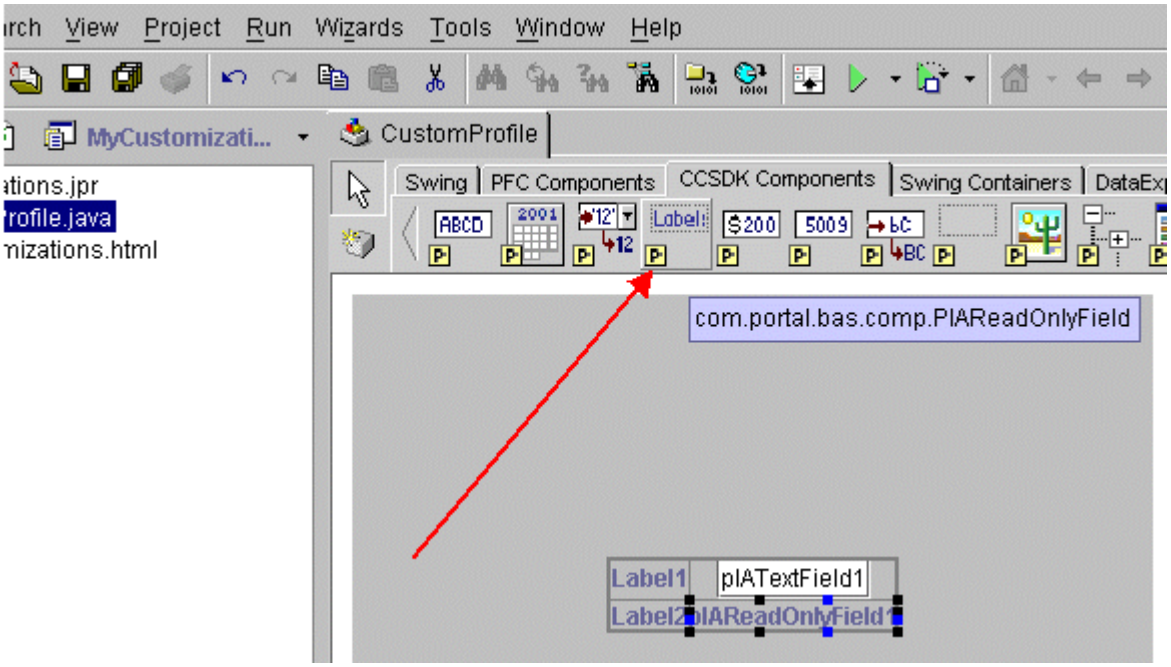
Figure 52-5 Adding a CCSDK PIATextField Widget



13. Drag a PIAReadOnlyField widget next to the second label widget.

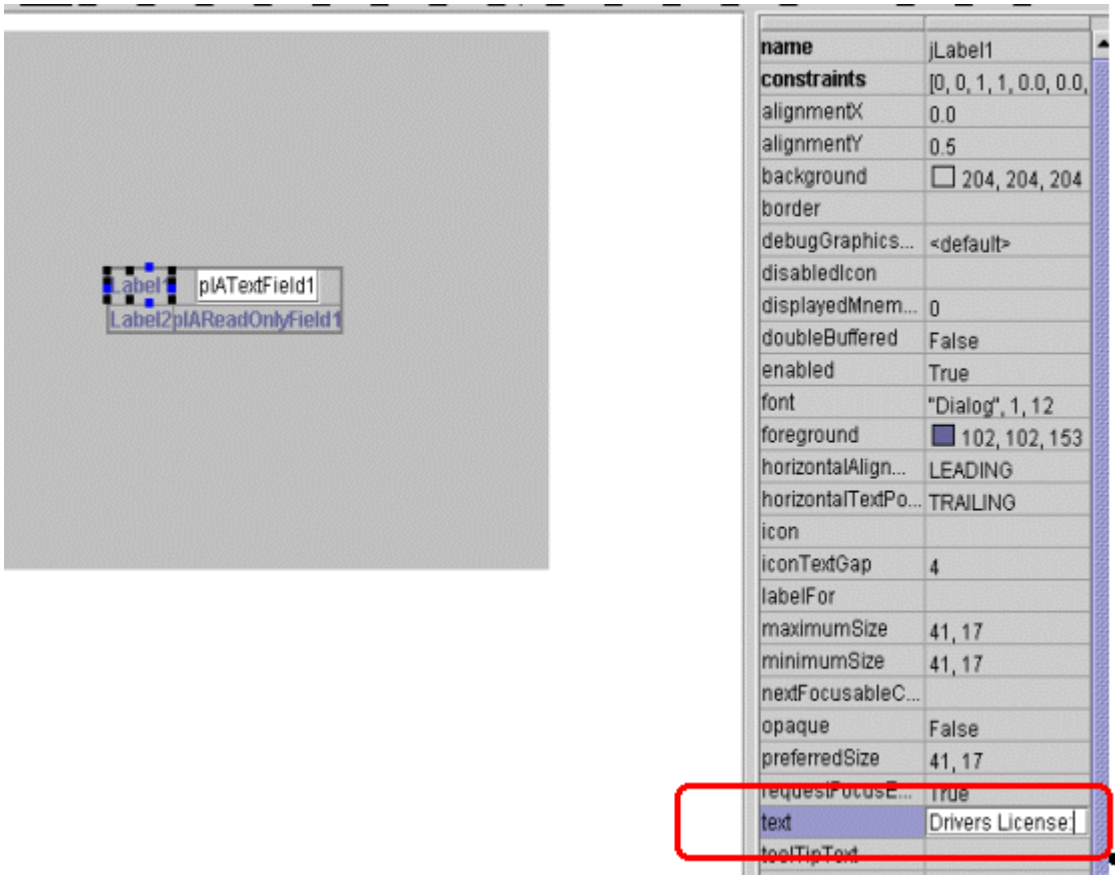
    In Figure 52-6, the first field is editable and the second is read-only:

Figure 52-6 Adding a PIAReadOnly Widget



14. Select the first label, click the text widget property, and enter the text Drivers License as shown in Figure 52-7:

Figure 52-7 Adding Text to a PIATextField



15. Click the second label and enter Credit Score: as its text property.
16. Click the first textfield widget (the PIATextField).

    In this example, you map this widget to the **/profile/creditscore** field that stores the drivers license data. With the widget selected, right-click the pIATextField1 instance variable on the left panel and choose **Customizer**.
17. To connect to BRM, enter the login and password.

> **Note:**
>
> You might need to expand the **Connection info** section and enter the host and port.

18. In the Customizer dialog box, scroll down, expand the **profile** section of the tree, and select the **/profile/creditscore** storable class. This is the profile object you created in Developer Center.
19. Select PIN_FLD_DN in the right pane.

    The information at the bottom of the Customizer is filled in. This is the mapping of BRM fields to Java field names, which maps your widget to a BRM field. The customizer automatically fills in the necessary information to enable BAS to read and write the data to and from BRM.

In this example, you can leave the DisplayFieldFormat as is, it is used only when you are working with a BRM field that contains data that does not make sense to a person, such as the int values that represent an account's status. See the SDK documentation for more information on this field.

20. Click the **Apply** button, and then click **OK**.

    The values are copied automatically from the Customizer to the appropriate widget properties.

21. Select the pIAReadOnlyField widget and display its customizer.

    You are already connected to BRM, so you do not need to log in again.

22. Select PIN_FLD_INT_VAL, click **Apply**, and then click **OK**.

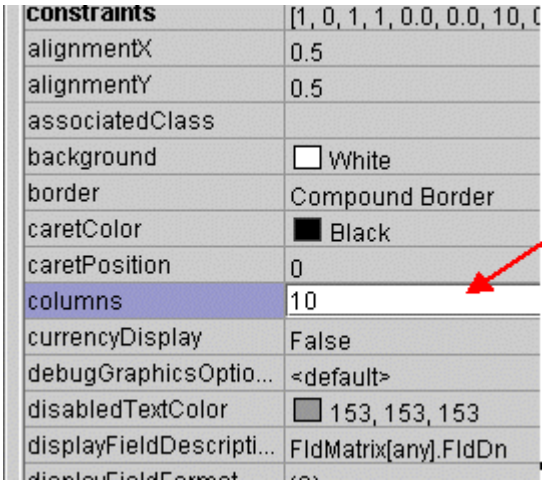
23. Save your changes in JBuilder.

    At this point, you are done. However, you can modify the layout by adding space between the widgets and renaming the widgets more appropriately. For example, to rename the textfield widget, select its name property and change it to driversLicenseTextField.

24. If required, modify the initial widget values.

    For example, select the textfield widget, select the columns property on the right side, and enter a value of **10** as shown in Figure 52-8.

Figure 52-8 Editing the Columns Property



25. Select the textfield widget and on the right side of the JBuilder work area, click the text property. By default, it contains pIATextField. Delete the entire string.

26. If required, modify the read-only widget.

    a. Right-click the widget and set its fill to Horizontal.

    b. Select its text property value and delete it.

> **Tip:**
>
> You can switch back to the source code window to see the generated JBuilder code.

27. Save your code.

28. Use Configurator to integrate your profile panel to Customer Center:
    - If you are adding the profile panel to the account maintenance interface, see "Tab Options".
    - If you are adding the profile panel to the new account interface, see "New Account Page Options".

What's Next

If you have no further Customer Center customizations to code, see "Building Your Customer Center Customizations".

53
Sample Customer Center Customizations

Learn how you can customize various fields and behaviors in several Oracle Communications Billing and Revenue Management (BRM) Customer Center tabs and in the Search feature.

Topics in this document:

- Building and Deploying Customizations
- Customizing Contact Fields
- Customizing Fields in the Balance Tab
- Customizing Fields in the Payments Tab
- Customizing Fields in the Services Tab
- Customizing Fields in the Hierarchy Tab
- Customizing Fields in the Sharing Tab
- Configuring Dynamic Drop-Down Lists

See also "Using Customer Center SDK" and "Customizing the Customer Center Interface".

Building and Deploying Customizations

To build and deploy the customizations described in this chapter, see "Building Your Customer Center Customizations" and "Deploying Your Customer Center Customizations".

Customizing Contact Fields

This section describes how you can customize the Customer Center contact fields. The contact fields appear in the Contact page of the New Account wizard and in the **Summary** tab.

> **Note:**
>
> For simplicity, this section uses **Contact** page to refer to both the Contact page in the New Account wizard and the contact fields in the summary tab.

Customizing Contact Fields

This section describes how to customize some default contact fields within a contact record. For information on modifying contact field behavior when there is more than one contact, see "Modifying Multiple Contact Behavior". For information on replacing address and contact panels with your own custom panels, see "Using Custom Address Panel and Contact Page".

Adding Drop-Down Lists to the Contact Type and Salutation Fields

To replace the **Contact type** and **Salutation** text fields with drop-down lists:

1. Extend the **com.portal.app.cc. PContactPage** class.
2. Call the following methods from the extended class:
   - For the **Salutation** drop-down list:

     ```
     protected final void setValidSalutations(String salutations[],String
     defaultSalutation)
     super.customizeNow();
     ```

   - For the **Contact type** drop-down list:

     ```
     protected final void setValidContactTypes(String contactTypes[],String
     defaultContactType)
     ```

   where:

   - The first parameter contains the list of values displayed in the drop-down list.
   - The second parameter specifies the default value for the field.

   Example method call:

   ```
   protected final void setValidSalutations(String
   salutations["Mr.","Ms.","Mrs."],String "Mr.")
   super.customizeNow();
   ```

See also "Disabling Changes to the Contact Type for the First Contact".

Populating Drop-Down List Values from a Properties File

To populate drop-down list values from a properties file:

1. Add the following line to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties** file:

   ```
   mycontactpage.contacttype=Billing,Mailing,Shipping
   ```

2. Add a method to your customized class, such as the one extended from **PContactPage**, that reads the drop-down list values from the properties file, as in this example for the **Contact type** field:

   ```
   private String[] getValidContactTypes() {
       try {
           final String delimeter = ",";
           Sring contacttypestring = getResourceBundle().getString("mycontactpage.contac
   ttype");
           return contacttypestring.split(delimeter);
       catch(MissingResourceException e){
       //
   A customized error message can be added here to informing the end user of the error
       String[] retval = {""};
       return retval;
   ```

   When the list of valid data is read, you can set it by calling the base class method:

   ```
   setValidContactTypes(getValidContactTypes(),"Billing");
   ```

Adding Drop-Down Lists to Address Panel Fields

To replace the **City**, **State/Province**, **ZIP/Postal**, and **Country** text fields in the **Contact** page with a drop-down list:

1. Extend the **com.portal.app.cc.PAddressPanel** class.
2. Call the following methods:

- For the **City** drop-down list:

  ```
  protected final void setValidCities(String[] cities,String defaultCity)
  ```

- For the **State/Province** drop-down list:

  ```
  protected final void setValidStates(String[] states,String defaultState)
  ```

- For the **ZIP/Postal** drop-down list:

  ```
  protected final void setValidZips(String[] zips,String defaultZip)
  ```

- For the **Country** drop-down list:

  ```
  protected final void setValidCountries(String[] countries,String defaultCountry)
  ```

where:

- The first parameter is the list of values displayed in the drop-down list.
- The second parameter specifies the default value for the field.

Populating drop-down list values from a properties file

To populate the drop-down list values from a properties file:

1. Add the **myaddresspanel** parameters to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties** file, as in this example for countries:

   ```
   myaddresspanel.countries=US,UK,India
   ```

   > **Note:**
   >
   > - This entry is for the country field. Follow the same procedure for the **City**, **State/Province**, and **ZIP/Postal** fields.
   > - The list drop-down list values will be displayed in the order placed in the parameter.

2. Add a method to your customized class, such as the one extended from **PAddressPanel**, that reads the drop-down list values from the properties file, as in this example:

   ```
   private String[] getValidCountries() {
       try {
           final String delimeter = ",";
           String countrystring = "A,B,C,D";
           return countrystring.split(delimeter);
         } catch (MissingResourceException e) {
           String[] retval = { "" };
           return retval;
           }
       }
   ```

   Once you have the list of valid data then you can set it by calling the base class method:

   ```
   String defaultCountry = "US";
   setValidCountries(getValidCountries(),defaultCountry);
   ```

Adding and Removing Item Listeners to Address Field Drop-Down Lists

You can add and remove item listeners to the **Address** drop-down fields that trigger actions depending on which value the CSR selects. For example, you can use listeners to dynamically

populate drop-down list values for the **State/Province** field depending on the country that the customer service representative (CSR) selects for the **Country** field.

To add and remove item listeners to the address fields, use these methods:

- For the **Country** drop-down list:

```
protected final void addCountryListener(ItemListener l)
protected final void removeCountryListener(ItemListener l)
```

- For the **State/Province** drop-down list:

```
protected final void addStateListener(ItemListener l)
protected final void removeStateListener(ItemListener l)
```

- For the **City** drop-down list:

```
protected final void addCityListener(ItemListener l)
protected final void removeCityListener(ItemListener l)
```

- For the **ZIP/Posta**l drop-down list:

```
protected final void addZipListener(ItemListener l)
protected final void removeZipListener(ItemListener l)
```

Modifying Multiple Contact Behavior

This section describes how to customize **Contact** page fields and behavior for accounts with more than one contact.

Specifying the Contact Type for Each Consecutive Contact

You can specify custom contact types by adding the following parameters to the **Customized.properties** file in the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory:

- For the first (mandatory) contact panel, use this parameter:

```
custinfo.panel.billingcontact.1=<customcontacttype>
```

- For the second and consecutive panels, use this property:

```
custinfo.panel.newcontact.index=<customcontacttype>
```

  where *index* is **2** for the second panel and increments by 1 for each subsequent panel. If a new contact has no corresponding type set, for example if you add a fourth contact and **custinfo.panel.newcontact indexes** are only available for contacts 1, 2, and 3, the contact type defaults to a blank field.

Disabling Changes to the Contact Type for the First Contact

You can disable the custom **Contact type** drop-down list for the first contact and restrict the field value to a certain value, such as **Billing**.

To set a static **Contact type** value for the first contact, call the following method with the parameter **true** from the contact type subclass:

```
protected final void setDisableBillingContactType(boolean b)
```

Configuring Duplicate Checking for the Contact Type Field

You can configure duplicate checking for the **Contact type** field by calling the following method with the parameter **true**:

```
protected final void setContactDuplicateCheckOn(boolean b)
```

By default, duplicate checking is turned off.

Configuring the Contact Type Validation Error Messages

You can customize the error messages that Customer Center displays when contact type uniqueness validation is enabled. (That is, when **setContactDuplicateCheckOn** is set to **true**).

Customer Center displays contact type error messages:

- When the contact type has already been specified for a previous contact and the CSR specifies a contact type for a secondary contact.

  By default, the error message displayed is **Contact type duplicate check is on, cannot be duplicated**. To change this message, add the following line to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties** file:

  custinfo.validate.duplicatecontacttypes=*Error_Message*

  where *Error_Message* is the error message string.

  > **Note:**
  >
  > Do not put the string in quotes.

- When the CSR attempts to add a new contact without selecting a valid contact type.

  By default, the error message displayed is **Contact type duplicate check is on, inadequate contact types**. To change this message, add the following line to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties** file:

  custinfo.validate.inadequatecontacttypes=*Error_Message*

  where *Error_Message* is the error message string.

  > **Note:**
  >
  > Do not put the string in quotes.

Using Custom Address Panel and Contact Page

This section describes how to replace the default Address panel and Contact page with your customized versions.

Replacing the Address Panel with a Custom Panel

> **Note:**
>
> This procedure only customizes the **City**, **State/Province**, **ZIP/Postal**, and **Country** fields. The **Address** field is not customized with this procedure.

To replace the **Address** panel of the **Contact** page with a custom **Address** panel:

1. Extend the **com.portal.app.cc.PAddressPanel** class.
2. Copy the Java source file of the custom class to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory.
3. Open the customized properties file (*CCSDK_home*/**CustomerCareSDK/CustCntr/custom/Customized.properties**) and add this line after the comment statements:

   ```
   com.portal.app.cc.PAddressPanel.subclass=MyAddressPanel
   ```

   where *MyAddressPanel* is the name of your extended class.
4. Save the file.

Replacing the Contact Page with a Custom Page

To replace the **Contact** page with a custom **Contact** page:

1. Extend the **com.portal.app.cc.PContactPage** class.
2. Copy the Java source file of the custom class to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom** directory.
3. Open the customized properties file (*CCSDK_home*/**CustomerCareSDK/CustCntr/custom/Customized.properties**) and add these lines after the comment statements:

   ```
   com.portal.app.cc.PContactPage.subclass=MyContactPage
   contactspage.class=MyContactPage
   helpid.acwizard.contactpage=MyContactPage
   ```

   where *MyContactPage* is the name of your extended class.
4. Save the file.

Customizing Fields in the Balance Tab

This section describes how to customize the **Action** drop-down list in the **Balance** tab.

Setting the Correct JRadioButtonMenuItem Button

To indicate the balance page state, you can add **JRadioButtonMenuItem** to the **Action** drop-down list on the **Bills** panel of the **Balance** tab:

1. Extend **PARBalancePage** class and create a **JRadioButtonMenuItem** object.
2. Add the object by calling the **addRadioMenuToAction(JRadioButtonMenuItem)** method.

Customizing Fields in the Payments Tab

This section describes how to customize various fields in the **Payments** tab.

Disabling the Billing Cycle & Tax Setup Link in the Payments tab

You can enable or disable the **Billing Cycle & Tax Setup** link in the **Payments** tab and enable or disable the fields in the **Tax Setup** panel.

- To enable or disable the **Billing Cycle & Tax Setup** link in the **Payments** tab, call the methods in Table 53-1:

Table 53-1 Payments Tab Methods

PPaymentPage methods	Description
disableBillingCycleAndTaxSetupLink ()	Disables the BillingCycleAndTaxSetup link
enableBillingCycleAndTaxSetupLink ()	Enables the BillingCycleAndTaxSetup link

- To enable or disable the fields in the **Tax Setup** panel, use the new **setTaxSetupEnabled(boolean)** flag. Call this method from the extended **PbillingCycleAndTaxSetupPage** class. This method is provided in **PbillingCycleAndTaxSetupPage**. The default is enable.

Configuring Values in the Billing Day of Month Combo Box

You can configure the list of available values in the **Billing day of month** spinner field in the **Payments - Billing Cycle & Tax Setup** panel. For example, you might want to offer a different billing day of month for each brand.

To configure the available values of the **Billing day of month** spinner field:

1. Extend the **PbillingCycleAndTaxSetupPage** class.
2. Create a custom controlled day of month widget by creating a **PIASpinnerField** object.
3. Assign a custom range of values to the object.
4. Call either the **setCustomDomFld(PIASpinnerField** *sf*) or the **setCustomDomComponent(Component sf)** method, where *sf* is the name of the custom widget.

> **Note:**
>
> - The **setCustomDomFld(PIASpinnerField)** method replaces the **Billing Day of Month** field **(PIASpinnerField)** with another custom **PIASpinnerField**.
> - The **setCustomDomComponent(Component)** method replaces the **Billing Day of Month** field **(PIASpinnerField)** with another custom **Component**.

5. In the Custom billing cycle/tax setup class field in the Configurator **Payments** tab, specify the custom **PbillingCycleAndTaxSetupPage** class name. See "Payment Configurator".

Setting the Next Billing Cycle Field to Visible or Not Visible

You can specify whether the **Next billing cycle** read-only field is displayed in the **Payment tab - Billing Cycle & Tax Setup - Billing Cycle** panel by using the **showNextBillingCycle(boolean)** method in an extended **PbillingCycleAndTaxSetupPage** class.

The **showNextBillingCycle(boolean)** method is included in **PbillingCycleAndTaxSetupPage.** This method can be used in the extended payment page to set the **Next billing cycle** field to visible or not visible.

To set the next billing cycle field to visible or not visible:

1. Extend the **com.portal.app.cc.PPaymentPage** class.
2. Call the **showNextBillingCycle(boolean)** method. Pass in **true** (default) for visible and **false** for not visible.

Customizing the Expiration Date Fields in the Credit Card Panel

You can replace the **Expiration date** field in the **Credit Card** panel of the **Payment Options** dialog box with a custom spinner field. To do so:

1. Open the customized properties file (*CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties**).
2. After the comment statements, add this line:

   ```
   paymentsetup.creditcard.expirationdate.usespinnerfields=true
   ```

3. (Optional) Set the year limits for the spinner field by adding these lines:

   ```
   paymentsetup.creditcard.expirationdate.spinnerfield.yearmax=Max_Limit
   paymentsetup.creditcard.expirationdate.spinnerfield.yearmin=Min_Limit
   ```

   where *Max_Limit* and *Min_Limit* are the upper and lower bounds for the year value.

   > **Note:**
   >
   > - *Max_Limit* must be greater than or equal to the *Min_Limit*. By default range for the year spinner field is **00** to **99**.
   > - If *Min_Limit* is not in the range of **00** to **99**, the system uses **00** for **yearmin**.
   > - If *Max_Limit* is not in the range of **00** and **99**, the system uses **99** for **yearmax**.
   > - The month spinner field is hard-coded with the range **00** to **12**. If the CSR selects the default **00**, an error is thrown and the CSR is prompted to select a correct month value.

4. Save the file.

Creating a Custom Payment Method

You can add a custom payment method to the **New Payment Method** drop-down list. This list appears on the Payment Options dialog box, which you can access from the **Payments** tab and the **Account Creation** wizard.

> **Note:**
>
> To implement custom payment methods, you must also add them to the BRM system so that your custom methods can be saved and retrieved. See "Adding a Custom Payment Method" in *BRM Opcode Guide*.

To create a custom payment method:

1. Create a custom panel that extends the BRM class **com.portal.app.cc.comp.PIAPaymentTypePanel**.

   The following sample code uses the name **NewPayPanel** for the custom panel:

   ```
   public class NewPayPanel extends com.portal.app.cc.comp.PIAPaymentTypePanel  {

   //used while acct creation
   public void shareInData() { }

   //used while acct maintenance
   public void shareInData(PModelHandle model) { }

   }
   ```

2. Create an interface that extends **com.portal.app.ccare.comp.PIAPaymentTypePanelBean**.

   The following sample code uses the name **NewPayBean** for the interface:

   ```
   public interface NewPayBean extends PIAPaymentTypePanelBean  {
   // NewPayBean is a blank interface
   //PIAPaymentTypePanelBean interface abstract methods are getting implemented in the
   controller class
   }
   ```

3. Create a controller class that extends **PIAComponentCollectionBean** and implements the interface you created in the previous step.

   The abstract methods in the **NewPayBean** interface need to be implemented in the new controller class for the custom payment method to work.

   In the following sample code, **NewPayBeanImpl** is the name of the new controller class:

   ```
   public class NewPayBeanImpl extends PIAComponentCollectionBean
                               implements NewPayBean   {

   //Appends additional create-time information onto the model passed in
   public void defaultsForStoring(PModelHandle model)
                               throws RemoteException {  }

   //Appends additional update-time information onto the model passed in
   public boolean defaultsForUpdate(PModelHandle model)
                               throws RemoteException {   }
   ```

```
//Appends additional validate-time information onto the model passed in
public boolean defaultsForValidation(PModelHandle model)
                              throws RemoteException {   }

//Gets the contact Name and Address Info  current account model
public NameAddressData getNameAddressData(PModelHandle modelHandle
                              throws RemoteException {   }

}
```

4. Add the following lines to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/Customized.properties** file:

```
consumerpayment.options=invoice creditcard ddebit new_pay
businesspayment.options=invoice creditcard ddebit new_pay
maintenance.options=invoice creditcard ddebit new_pay
new_pay.selector=id
id.class=NewPayPanel
```

where:

- *new_pay* is a text string that identifies your custom payment method internally. It is not the text that will be displayed in Customer Center.
- *id* is the ID for the new payment method. The ID must be a 5-digit number starting with 100. 10000 to 10016 are already used in the default version of Customer Center. Do not use the same number for more than one payment method.
- *NewPayPanel* is the name of the customized panel class for the new payment method.

> **Note:**
>
> The **invoice**, **creditcard**, and **ddebit** strings represent the default payment methods.

5. Save and close **Customized.properties**.
6. Add the following line to the *CCSDK_home*/**CustomerCareSDK/CustCntr/custom/CustomizedResources.properties** file:

```
payType.format={0,choice,0#Unknown|10000#Prepaid|10001#Invoice|10002#Debit|
10003#Credit Card|10004#Direct Debit (Fr)|10005#Direct Debit|10006#Smart Card|
10007#Nonpaying child|10008#Unknown|10009#Undefined|10010#Guest|10011#Cash|
10012#Check|10013#Wire-Transer|10014#Inter Bank Payment order|10015#Postal order|
10016#Voucher|id#New_Pay_Label}
methodofpayment.id=New_Pay_Label
```

where:

- *id* is the 5-digit number you entered in **Customized.properties**.
- *New_Pay_Label* is the name displayed in Customer Center for the new payment method.

7. Save and close **CustomizedResources.properties**.

Customizing Fields in the Services Tab

This section describes how to customize various fields in the **Services** tab.

Adding Charges for SIM and MSISDN Changes

To charge for Subscriber Identity Module (SIM) changes and Mobile Station International Subscriber Directory Number (MSISDN) changes:

1. Do one or both of the following:
   - To charge for SIM changes, subclass the **SIMPanel.java** class.
   - To charge for MSISDN changes, subclass **NUMPanel.java**.
2. Create a customized panel with the necessary fields and pass them to the **setCustomData(PIACustomizablePanel)** method as an argument.

   The new panel appears at the end of the existing SIM or Number panels in the **Services** tab.
3. In the Configurator **Service** tab, specify the custom class names in the Custom SIM Panel Class and Custom Number Panel Class fields. See "Service Configurator".

Adding a Secondary MSISDN for Supplementary Services

To add a secondary MSISDN for supplementary services:

1. Create a custom **PPurchaseOfferingAction** class by extending it.
2. Specify the class path and key in the **customized.properties** file, as in this example:

   `customized.PurchaseOfferingAction.class=<com.portal.app.cc.TestOfferingAction>`
3. Create a custom **PPurchaseOfferingWizard** class by extending it.
4. Specify the class path and key in the **customized.properties** file, as in this example:

   `customized.PurchaseOfferingWizard.class=<com.portal.app.cc.TestOfferingWizard>`
5. Create a custom search entry panel class (**PTelcoNumberEntryPanel)** by extending it.
6. Specify the class path and key in the **customized.properties** file, as in this example:

   `device.num.search.entry.panel.class=<com.portal.app.cc.tcf.TestNumberEntryPanel >`
7. Create a custom search results panel class (**PTelcoNumberResultPanel)** by extending it.
8. Specify the class path and key in the **customized.properties** file, as in this example:

   `device.num.search.results.panel.class=<com.portal.app.cc.tcf.TestNumberResultsPanel>`
9. Create a custom purchase offering wizard by extending the **PPurchaseOfferingWizard** class.
10. In the extended class, override the **protected Object commitData(PModelHandle model) throws RemoteException** method.
11. (Optional) Create a search popup dialog box for the secondary MSISDN number based on the services that are purchased in the current **MdelHandle**.

To call the custom opcode in **commitData**, call your own method. If you do not want to call the custom opcode, call **super.commitData(model)**.

To retrieve the default value in the **Number Category combo** field, use the **public void setDefaultToNumCategory(String defaultStr**) method in the **PTelcoNumberEntryPanel** class. This can be called with a string (the default value) as a parameter for setting default values in that field.

Customizing Fields in the Hierarchy Tab

This section describes how to customize various fields in the **Hierarchy** tab.

Adding a Custom Popup Component to the No Hierarchy Page

You can use the **addAdditionalActions(AbstractAction[] actions)** method in **PAcctNoHierarchyPage** to add a custom popup component to the title panel. When this method is called from the **PAcctNoHierarchyPage** subclass, a new action drop-down is displayed the first time the method is called. If the method has been previously called, the action is added to the existing drop-down list.

To set a label for the action:

1. Define the **public String getMenuLabel()** method with a return string as the label value.
2. Implement the action event by defining the **public void actionPerformed(ActionEvent)** method.

This sample code describes a **PAcctNoHierarchyPage** subclass:

```
public class MyHierarchyPage extends PAcctNoHierarchyPage {
    public MyHierarchyPage() {
        PAddOnAction[] actions = new PAddOnAction[1];
        actions[0] = new MyActionA();
        addAdditionalActions(actions);
    }
}
class MyActionA extends PAddOnAction {
    public MyActionA() {
    }
        public String getMenuLabel() {
    return "Action A";
    }
public void actionPerformed(ActionEvent e){
        //Custom Action
        //JOptionPane.showMessageDialog(null,"My Action A");
    }
}
```

Adding a Custom NoHierarchy Page

To replace the default NoHierarchy page with a custom NoHierarchy page by using Configurator:

1. Create a custom NoHierarchy page class.
2. Start Configurator.
3. Go to the Custom NoHierarchy Page Class field on the Configurator **Hierarchy** tab. See "Hierarchy Configurator".
4. Type the full path of your customized page class name in this field to replace the default No-Hierarchy-Page, as in this example:

   ```
   com.myComp.app.CustomNoHierarchyPage
   ```

Creating Customized Search Dialogs and Disabling the To Field

You can create your own search dialog box and disable the **To** field in the **Move account** panel in the **Hierarchy** tab.

1. Extend the **com.portal.app.cc.PhierarchyMovePage.java** class and override the **search_actionPerformed(ActionEvent e)** method to launch a custom action to call a custom Search dialog box.

   > **Tip:**
   >
   > See the sample code **MyHierarchyMovePage.java**.

2. Disable or enable the **To** field in the **Move account** panel by setting the **setEnabledToField(boolean) boolean** method.
3. In the Custom Hierarchy Move Page Class field in the Configurator **Payments** tab, specify the custom payment setup class name. See "Hierarchy Configurator".

Adding Custom Options to the Actions Drop-Down Lists

The Customer Center SDK includes the methods in Table 53-2 for adding custom options to **Action** drop-down lists in a hierarchy page:

Table 53-2 Custom Options Methods

Method	Description
public void treeValueChanged(TreeSelectionEvent)	Triggers events when items are selected in a hierarchy tree.
public void addAdditionalActions(AbstractAction[])	Accepts the **PBASAction** and **PAddOnAction** arrays as arguments.

To add custom options to the Action drop-down lists:

1. Create a custom action class by extending the **PAddOnAction** class and overriding these methods:
   - **public String getMenuLabel()**
   - **public void actionPerformed(ActionEvent e)**
   - **public void treeValueChanged(TreeSelectionEvent e)**

   > **Note:**
   >
   > Override this method to generate an event whenever a new item is selected in the hierarchy.

2. Write a custom Account Hierarchy page by extending the **PAcctHierarchyPage** class and using the **addAdditionalActions** method to add your action to the **Action** drop-down menu.

See the sample code in **MyHierarchyPage.java**.

Customizing Fields in the Sharing Tab

This section describes how to customize fields in the **Sharing** tab.

Adding a New Sharing Type to the View Drop-Down List

You can create your own sharing types and create supporting dialog boxes. To create a sharing type, you must define the sharing type, its **Sharing** panel and all other dialog boxes, its controller class, a unique ID, and a unique label. See "Customizing the Customer Center Interface".

You then add the new sharing type to the **View** drop-down on the **Sharing** tab. To do this, you modify the **Customized.properties** file to add the new **Sharing** panel class, controller, and option name used internally. You also modify the **CustomizedResources.properties** file to add the label you want displayed in the drop-down. You then package these files in the **ccCustom.jar** file, which should be added to the file you use to run customer center (**runCC.bat**, for example).

To modify the **Customized.properties** file and **CustomizedResources.properties** files:

> **Note:**
>
> These files are located in the *CustomerCareSDK_home*/**CustomerCareSDK/CustCntr/custom** directory.

1. Open the **Customized.properties** file in a text editor and add these lines for each new sharing type:

   ```
   customercenter.sharing.NewSharingType.class = ClassName
   customercenter.sharing.NewSharingType.controller = ControllerName
   ```

   > **Note:**
   >
   > Be sure to use fully qualified class and controller names. If there are multiple sharing options in the last line, delimit them with commas.

2. Add the following line:

   ```
   customercenter.sharing.options = PCharge, PDiscount,NewSharingType
   ```

   This line appears only once in the **Customized.properties** file and should include the option names of the standard BRM sharing types (PCharge and PDiscount) as well as the option name of each new sharing type you include in the file. The order of the list determines the order of the options in the **View** box on the Customer Center **Sharing** tab.

3. Save the file.

4. Open the **CustomizedResources.properties** file in a text editor and add this line:

   ```
   customercenter.sharing.NewSharingType.label = StringName
   ```

   The string name is the sharing type name you want to appear in the **View** drop-down.

5. Save the file.
6. (Optional) On the Customer Center SDK **Sharing** tab, select the new group on the **Order the Sharing combo box** list and click **Raise Order** or **Lower Order** to rearrange the drop-down list.

   For information on the Customer Center SDK **Sharing** tab, see "Sharing Configurator".

The following samples show a customization that adds a new sharing group for free Megabytes:

Customized.properties file

```
customercenter.sharing.PMbyte.class = com.portal.app.cc.sharing.PMbyteSharingPanel
customercenter.sharing.PMbyte.controller =
com.portal.app.cc.sharing.PMbyteSharingController

customercenter.sharing.options = PCharge, PDiscount, PMbyte
```

CustomizedResources.properties file

```
customercenter.sharing.PMbytes.label = Megabyte Sharing
```

Configuring Dynamic Drop-Down Lists

To configure a dynamic drop-down list whose labels and values are based on data stored in BRM:

1. Create a custom properties object by creating a new class that implements the new **LoadCustomProperties** interface and providing the implementation for the **public Properties loadCustomProperties()** method. You must override this method with your custom code that accumulates properties (name/value pairs).
2. In the Class name for loading the Custom Properties field in the **Other** tab in Configurator, specify the class name that implements the **LoadCustomProperties** interface. See "Other Settings".

> **Note:**
>
> You must define the fully qualified class name, for example, **com.helloworld.MyInterface**.

See the sample code in the **TestInterface.java** file.

Part VII
Localizing BRM

This part describes how to localize Oracle Communications Billing and Revenue Management (BRM). It contains the following chapters:

- Using BRM in International Markets
- BRM Internationalization and Localization
- Creating a Localized Version of BRM
- Handling Non-ASCII Code on the BRM Server

54

Using BRM in International Markets

Learn how Oracle Communications Billing and Revenue Management (BRM) supports customer management and billing for international markets.

Topics in this document:

- Supporting Multiple Currencies
- Accepting Credit Card Payments in Multiple Currencies
- Supporting Multiple Languages
- Using Localized Client Applications
- Localizing BRM

See also "BRM Internationalization and Localization" and "Creating a Localized Version of BRM".

Supporting Multiple Currencies

If you have customers in more than one country, there are two currency issues that you might need to work with:

- Customers who use a currency different from the currency you use in your business. For example, if your business is in the United States, you run your business with US dollars. However, your Canadian customers pay their bills with Canadian dollars. To handle multiple currencies, BRM uses a *system currency* for your business, and *account currencies* for your customers. See "Managing System and Account Currencies" in *BRM Managing Customers*.
- Customers in EMU countries who recently joined the EMU, and are still in the currency crossover period. They can still pay their bills in euros or in their native EMU country currency. BRM allows you to use both EMU currencies and the euro. See "Supporting EMU Currencies and the Euro" in *BRM Managing Customers*.

> **Note:**
>
> For countries that joined the EMU before February, 2002, the euro is the only legal currency.

Accepting Credit Card Payments in Multiple Currencies

The ability to accept credit card payments in multiple currencies depends on your credit card processor. For information, see "Paymentech and International Transactions" in *BRM Configuring and Collecting Payments*.

Supporting Multiple Languages

You need to handle the following situations when you have customers that use multiple languages:

- Your customers need to receive messages and invoices from you in their language. To do so, you specify the customer's language when creating the account.
- Your customers might need to access Web pages in their native languages. You can create multiple sets of Web pages, each in a localized language. When a customer logs in, the customer's language setting automatically opens the Web page in the correct language.
- Your BRM system, including your BRM database, needs to handle multiple types of characters, for example, Japanese and Chinese characters and characters with accent marks.

Using Localized Client Applications

Localized versions of BRM client applications are available in the following languages:

- Brazilian
- Chinese Simplified
- Chinese Traditional
- French
- Italian
- Japanese
- Korean
- Portuguese
- Russian
- Spanish

You can use localized versions of many BRM client applications:

- All language versions of the Java applications use Unicode characters.

All BRM client applications are internationalized and handle most locale formats for time, date, number, and so on by using the Windows Regional Settings. To keep currency symbols constant whatever the locale, they are not handled by Windows Regional Settings.

See your sales representative for the latest information on available localizations. For client platforms supported, see "BRM Software Compatibility" in *BRM Compatibility Matrix*.

Localizing BRM

You can localize BRM in these ways:

- If a localized version of a BRM application does not already exist, you can create localized versions in additional languages using the Localization SDK.
- You can localize some system files, such as reason codes, by using the Localization SDK.
- You can create localized versions of your custom client applications.

The Localization SDK supports localization into any language that is both noncomplex text and single-direction (left-to-right), including languages written with the Roman alphabet and multi-byte or East Asian languages.

BRM does not support localization into complex text languages, including Thai, Indic languages, and languages that use bi-directional writing systems, such as Arabic and Hebrew.

55

BRM Internationalization and Localization

Learn about internationalization and localization issues for Oracle Communications Billing and Revenue Management (BRM) developers.

Topics in this document:

- About Localizing and Internationalizing
- About Internationalization of BRM Client Applications
- Writing Localized MFC Client Applications
- About Internationalized Development on BRM

About Localizing and Internationalizing

Localization and internationalization are related but not identical:

- **Localization (L10N)** is adapting a software product for a specific market (locale). This requires the following procedures:
  - Translating the product interface and help files into the locale language
  - Supporting the date, time, number, currency formats, and collation order of the locale
  - Supporting the input methods of the language
  - Possibly changing the content of the application, depending on the product and market
- **Internationalization (I18N)** is a process of developing software products that are independent from cultural, language, or other specific attributes of a market and can be easily localized.

  This includes designing user interfaces to handle languages that need more space, placing text strings in a resource file instead of hard-coding them, and using icons that have meaning across cultures.

About Internationalization of BRM Client Applications

BRM client applications are internationalized to work with languages using text that is both noncomplex and single-direction (left-to-right), including most languages of Western European and East Asian origin.

For European languages, the client applications support any Windows Regional setting locale that uses code page 1252. These are languages of Western European origin or languages that use a very similar alphabet including Afrikaans, Basque, Catalan, Dutch (standard), and Dutch (Belgian).

For East Asian multibyte locales, the client applications support Japanese (code page 932) Korean (949), Simplified Chinese (936), and Traditional Chinese (950).

Writing Localized MFC Client Applications

To use the Windows Regional Settings for a locale, you must follow the Microsoft Developer Network standards. Some of the most important APIs are those for:

- String manipulation
- Locale-related APIs, such as **GetLocaleInfo** and **enum**, and **LC**_* types for currency, date, and so on
- Code pages

About Internationalized Development on BRM

For international development, text strings must be isolated from other parts of the software. For information about:

- Using the proper conventions for storing text strings, see "String Manipulation Functions" in *BRM Developer's Reference*.
- Storing non-ASCII text, see "Handling Non-ASCII Code on the BRM Server".
- Loading your strings into the database, see "load_localized_strings".

56

Creating a Localized Version of BRM

Learn how to use the Oracle Communications Billing and Revenue Management (BRM) Localization Software Developers Kit (SDK) to create localized versions of BRM client applications and Self-Care Manager.

Topics in this document:

- About the Localization SDK
- System Requirements for the Localization SDK
- Building the Clients
- Packaging Your BRM Client Localizations
- Modifying Localized Versions of Customer Center
- Localizing Self-Care Manager
- Localizing and Customizing Strings
- Localizing BRM Reports
- About Customizing Server Software
- Locale Names

About the Localization SDK

The Localization SDK is for:

- Localization agencies who make the localized versions of BRM.
- Customers who develop localized versions of BRM client applications for languages that are not supported by BRM.

You can use the SDK to:

- Translate menus, dialog boxes, and online Help for BRM clients, including those written using Microsoft Foundation Class (MFC) and those written in Java.
- Make minor customizations of Customer Center that do not involve translation. For example, you can change text in the user interface or replace a bitmap image. See "Modifying Localized Versions of Customer Center" for more information.

  The Localization SDK is available in the same language versions as BRM client applications, so you can make these types of customizations to Customer Center in all available languages. For a list of supported languages, see "Localizations Supported".

> **Note:**
>
> You cannot translate the installation screens for an application. When you create an application with the SDK, the application's installation screens use the same language as the version of the SDK you used.
>
> For example, if you use the English SDK to translate an application into another language, the installation for the translated application is in English. If you use the French SDK to modify a French version of an application, the installation is in French.

Localization SDK Contents

This section describes localization SDK contents.

- Java Client Applications
- Self-Care Manager Server Application
- BRM Server Files

Java Client Applications

The Localization SDK contains all the Java properties files, Help files, and installation files that need to be translated for the following BRM applications:

- Business Configuration Center, which includes Field Validation Editor
- Customer Center, which includes Event Browser
- GSM Customer Center Extension
- Number Administration Center
- Payment Center
- Permissioning Center
- Pricing Center, which includes Resource Editor and Zone Mapper
- Revenue Assurance Center
- SIM Administration Center
- Suspense Management Center
- Voucher Administration Center

Self-Care Manager Server Application

The Localization SDK contains all the Java properties files, Java server pages, and installation files that need to be translated for Self-Care Manager, the browser-based self-care application.

For information about localizing Self-Care Manager, see "Localizing Self-Care Manager".

BRM Server Files

The Localization SDK also contains these BRM server files:

- Other server files containing text used by client applications. See "Localizing and Customizing Strings".

Localizations Supported

BRM supports localization into any language that is both non-complex text and single-direction (left-to-right), including:

- Languages written with the Roman alphabet, including Western European languages.
- Other European languages, including Russian and Greek.
- Multi-byte or East Asian languages, such as Korean, Japanese, Traditional Chinese, and Simplified Chinese.

> **Note:**
>
> Creating localized tools can represent a significant customization effort, especially for certain languages, including Norwegian, Danish, Russian, and Greek.

BRM does not support localization into complex text languages, including:

- Complex text languages that use bi-directional writing systems, such as Arabic and Hebrew.
- Other complex text languages, such as Thai and Indic languages.

System Requirements for the Localization SDK

To use the SDK, you need the following minimum requirements:

- System hardware:
  - 256 MB RAM
  - 500 MB of disk space for your build tree and zip files, preferably on one dedicated drive
- The client application you want to localize.
- Tools

  You need some or all of these tools, depending on the files you are localizing. For many of these tools, you can find the version compatible with the current BRM release in *BRM Compatibility Matrix*.
  - Microsoft Visual C++ plus the special files for your locale
  - Microsoft HTML Help Workshop or a Help authoring tool compatible with Microsoft HTML Help
  - Java 2 SDK, Standard Edition, international version
  - Java Runtime Environment (JRE), international version
  - MKS Toolkit
  - Command-line version of the zip file utility PKZIP

> **Note:**
>
> Before using the SDK, install all the applications listed in their default locations on the local drive of the source build machine.

To test your translated applications, you also need to run them on Windows in the target locale. That computer also needs to have, or have network access to, a machine with a complete BRM installation.

Building the Clients

This section describes how to create localized versions of BRM Java client applications. For information on localizing the Self-Care Manager server application, see "Localizing Self-Care Manager".

To build a Java client application, do the following:

1. Localize the BRM client application's properties files.
2. Optionally, add custom currency symbols to the BRM client application. See "Changing the Currency Format".
3. Create a JAR file of your translated properties files. See "Building Properties Files"
4. (Customer Center Only) Perform additional steps for Customer Center. See "Preparing Customer Center".
5. Add your localized files to a BRM client application. See "Packaging Your BRM Client Localizations".

Changing the Currency Format

To add a currency format into BRM Clients Application, add the following lines to the **CustomizedResources.properties** file:

resourceID=*currencySymbol*:*currencyCode*:*groupSep*:*decSep*:*posFmt*;*[negFmt]*

where:

- *resourceID* is the unique ID of the currency resource.
- *currencySymbol* is the character that represents the currency, for example, $ or €.
- *currencyCode* is the three-letter code for the currency.
- *groupSep* is the group separator, for example, the character to use between the numbers for thousands and hundreds.
- *decSep* is the separator to use between the whole and decimal parts of the amount.
- *posFmt* is the format to indicate positive numbers.
- *negFmt* is the optional format to indicate negative numbers.

For example:

```
840=$:USD:,:.:M#,##0.00;(M#,##0.00)
```

Building Properties Files

You can create all the JAR files with the same command or create them individually.

To create a JAR file of your translated properties files:

1. At a Windows command prompt, go to the **JAVA_PROJECTS** directory.
2. Set up an environment variable:

   ```
   set CLIENTS_DEST=W:\Client_dest
   ```

3. Go to **W:***SDK_locale***JAVA_PROJECTS** and run **makecertificate.bat**.

   where *SDK_locale* is the Windows locale for the version of the Localization SDK you installed.

   You can do this from the command line or through Windows Explorer.

   When you run **makecertificate.bat**, you create a keystore password and a key password.
4. Enter the passwords you created when running **makecertificate.bat** into the **signjar.bat** file, located in the same folder:
   - Enter the keystore password for STOREPASSWORD.
   - Enter the key password for KEYPASSWORD.
5. Run the **makeResourceJars** command:

   ```
   W:\SDK_locale\JAVA_PROJECTS> makeResourceJars Java_locale [app_name]
   ```

   where
   - *Java_locale* is the two-letter ISO 639 language code.
   - *app_name* is the abbreviated name of an application; for example, **ebrowser**. Specify this only if you are building a JAR file for a single application. Table 56-1 shows the names of JAR files for applications.

Table 56-1 Names of JAR Files for Applications

Application	Name for makeResourceJars
Event Browser	ebrowser
Invoice Viewer	invoice
Pricing Center	price
Customer Center	cutscent
Config Center	bmf
Field Validation Editor (Config Center)	fve
Resource Editor (Config Center)	re
Bulk Accounts (Config Center IPT)	bat
Zone Mapper (Config Center)	zmp
WebKit	webkit
GSM Manager Client (Customer Center Add-on)	gsm
SIM Administrator	sim
Number Administrator	num

If a filename is not specified, this command builds a locale-specific JAR file for each application and puts the JAR files in the **Client_dest** directory.

If a filename is specified, the command builds a single locale-specific JAR file and places it in the **Client_dest** directory.

For example, to create a JAR file for a French localization of Event Browser, enter the following command:

```
W:\SDK_locale\JAVA_PROJECTS> makeResourceJars fr ebrowser
```

> **Tip:**
>
> Enter **makeResourceJars** with no arguments to see its syntax.

Preparing Customer Center

Customer Center requires a few additional steps before packaging.

1. Copy the **CustomerCenter_en.html** file to **CustomerCenter**_*Java_locale*.**html**, where *Java_locale* is the Java name for the new locale. For example, if creating a French version, copy the file to **CustomerCenter_fr.html**.
2. Translate the localizable strings in **CustomerCenter**_*Java_locale*.**html**.
3. Change all instances of **_en** in the file to _*Java_locale*. For example, for a French version, change **_en** to **_fr**.
4. Save and close **CustomerCenter**_*Java_locale*.**html**.
5. Copy **CustomerCenter_en.jnlp** to **CustomerCenter**_*Java_locale*.**jnlp**; for example, **CustomerCenter_fr.jnlp**.
6. Repeat earlier step 2 and 3 for **CustomerCenter**_*Java_locale*.**jnlp** to translate the localizable strings and change **_en** to _*Java_locale*.
7. Save and close **CustomerCenter**_*Java_locale*.**jnlp**.

Packaging Your BRM Client Localizations

To add the localized client files to a BRM client application:

1. Verify that you have the W drive mapped to **C:\Program Files\Portal Software\Localization SDK**.
2. Create the **W:***SDK_locale***zips** directory.
3. Download the zip file for the BRM application.
4. In a command window, go to the **W:***SDK_locale***bin** directory.
5. Enter the following command, replacing the variables with the actual package name and local names.

   ```
   W:\SDK_locale\bin> package_name Java_locale Windows_locale
   ```

   For example, to package a German version of Pricing Center, enter:

   ```
   W:\SDK_locale\bin> Package_PricingCenter.bat de deu
   ```

   This creates a zip file in the **zips** directory. This zip file has the same name as the original application zip file, but with the Java locale added.

   See "Locale Names" for a list of Java and Windows locales.

   The following BRM packages are available.

   - **Package_ConfigurationCenter.bat** packages Field Validation Editor and Configuration Center.
   - **Package_CustomerCenter.bat** packages Customer Center and Event Browser.

- **Package_GSMMgrClient.bat** packages the GSM Customer Center Extension.
- **Package_NumAdmin.bat** packages Number Administration Center.
- **Package_PaymentCenter.bat** packages Payment Center.
- **Package_PermissionCenter.bat** packages Permissioning Center.
- **Package_PricingCenter.bat** packages Pricing Center, Resource Editor, and Zone Mapper.
- **Package_SIMAdmin.bat** packages SIM Administration Center.
- **Package_RevenueAssurance.bat** packages Revenue Assurance Center.
- **Package_SuspenseManagement.bat** packages Suspense Management Center.
- **Package_VoucherAdministration.bat** packages Voucher Administration Center.
- **Package_WebKit.bat** packages Self-Care Manager (see "Localizing Self-Care Manager" for details on Self-Care Manager).

> **Note:**
>
> These packages assume you are using the English Localization SDK. If you are using a different version of the SDK, you need to edit these **.bat** files to add the correct Windows locale to the name of the zip files. For example, if you are using the French SDK, change **7.5_PaymentTool.zip** to **7.5_PaymentTool_FRA.zip**.

Modifying Localized Versions of Customer Center

This section explains how to use the BRM Localization SDK or the Customer Center SDK to customize localized versions of Customer Center.

About Simple Customization

The Localization SDK is available in the same language versions as BRM client applications. If you use a language version of Customer Center that BRM supports and want to make minor modifications, such as changing the original localized translation or changing the date format, use the Localization SDK in that language. See "Simple Customization for Localized Versions of Customer Center".

The Customer Center SDK is a superset of the Localization SDK so you can also use the Customer Center SDK to make these minor changes, but it is recommended that you use the Localization SDK in your language for this purpose.

> **Note:**
>
> The Customer Center SDK is available only in English.

About Advanced Customization

If you want to make extensive modifications to Customer Center, such as adding or removing a localized string or tab, you must use the Customer Center SDK. To use the Customer Center SDK to make extensive modifications to Customer Center, see "Advanced Customization for Localized Versions of Customer Center".

You may need to use both the Customer Center SDK and the Localization SDK for advanced customization. See "When to Use the Localization SDK for Advanced Customization".

Before You Begin

Before you customize the localized versions of Customer Center, you should be familiar with the following topics:

- Using Customer Center SDK
- Customizing the Customer Center Interface

> **Note:**
>
> The property files for BRM Java applications are in Unicode.

Simple Customization for Localized Versions of Customer Center

Use this procedure to make simple customizations to localized versions of Customer Center, such as modifying an original translation or changing the date format, by using the Localization SDK. For more information, see "About Simple Customization".

If you want to remove or add strings or tabs in Customer Center, see "About Advanced Customization" and "Advanced Customization for Localized Versions of Customer Center".

1. Install Customer Center for the language you will customize. See *BRM Installation Guide*.
2. Install the Localization SDK in the language you will customize.

   See the following:

   - System Requirements for the Localization SDK
   - Installing the Localization SDK on Windows in *BRM Installation Guide*.
3. Convert the property file from Unicode to the encoding that supports your locale.
4. Find what you want to change, such as the original translation or date format, in the Localization SDK.
5. Change it.
6. Convert the property file back to Unicode.
7. Run the **makeResourceJars** batch file to generate a signed **CustomerCenter**_*Java_locale*.**jar** file.

   For more information, see the **readme.txt** file in the Localization SDK installation.
8. Deploy your custom **CustomerCenter**_*Java_locale*.**jar** file.

   See "Deploying a Simple Customization of Customer Center".

Deploying a Simple Customization of Customer Center

If you are using a WebStart installation of Customer Center, use this procedure to deploy your custom localized **CustomerCenter**_*Java_locale*.**jar** file.

1. Go to the directory where you installed Customer Center on your Web server, *Customer_Center_home*.
2. Create a new folder named **custom** in *Customer_Center_home*.
3. Copy the signed **CustomerCenter**_*Java_locale*.**jar** file that you generated using the Localization SDK to the *Customer_Center_home***custom** folder.
4. Copy and paste the following text into the file *Customer_Center_home***custom\custom.jnlp**:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for Custom jar files -->
<jnlp
      spec="1.0+"
      codebase="http://your_server"
      href="custom/custom.jnlp">

      <information>
           <title>Custom</title>
           <vendor>XYZ, Inc.</vendor>
      </information>

      <security>
           <!-- <all-permissions/> -->
      </security>

      <resources>
           <j2se version="1.4*"/>
           <jar href="custom/CustomerCenter_Java_locale.jar"/>

      </resources>

      <component-desc/>

</jnlp>
```

5. Uncomment the following line in the **CustomerCenter**_*Java_locale*.**jnlp** file by removing the **<!--** and **-->** characters shown here:

```
<!-- <extension name="Customized" href="custom/custom.jnlp"/> -->
```

6. In the same file, remove the reference to the original **CustomerCenter**_*Java_locale*.**jar**, by making it an HTML comment:

```
<!-- <jar href="lib/CustomerCenter_Java_locale.jar"/> -->
```

7. Save the **custom.jnlp** file.
8. Restart Customer Center.

Advanced Customization for Localized Versions of Customer Center

Use this procedure to customize a non-English version of Customer Center extensively, such as renaming or removing fields or adding entirely new fields, by using the Customer Center SDK. For more information, see "About Advanced Customization".

In some cases, you may also need to use the Localization SDK. See "When to Use the Localization SDK for Advanced Customization".

> **Tip:**
>
> If you use a language that BRM supports and want to make minor modifications to Customer Center, such as changing some translations, see "About Simple Customization" and "Simple Customization for Localized Versions of Customer Center".

1. Install Customer Center for the language you will customize. See "Installing Customer Center" in *BRM Installation Guide*.
2. Install the Customer Center SDK. See "Installing BRM Thick Clients" in *BRM Installation Guide*.

   > **Note:**
   >
   > The Customer Center SDK is available only in English.

3. Go to the Customer Center SDK install directory (*CCSDK_home*).
4. Find the resources used in Customer Center in this file:

   *ccsdkinstall***CustCntr\Settings\CustomerCenterResources.properties**
5. Copy the *key* for the string you want to modify. These string entries use this syntax:

   `custinfo.service.label``=`*`value`*
6. Go to the *CCSDK_home***CustCntr\custom** directory.
7. Paste the key you copied into the properties file **CustomizedResources.properties**, and add your custom *value* next to the key, separated by an equal sign (**=**).

   For example:

   `custinfo.service.label``=`*`customized label value`*
8. To add new key-value pairs or to enter new values for existing keys in Customer Center, repeat steps 5 through 7 for copying the *key*, going to the *CCSDK_home***CustCntr\custom** directory, pasting the key into the properties file and adding your custom value.
9. When you finish editing this file, run the *CCSDK_home***buildAll.bat** script with the **CustCntr** parameter.

   This builds the **ccCustom.jar** file that contains **CustomizedResources.properties** and any other customizations. This script also signs the **ccCustom.jar** JAR file with your own certificate and copies the results to the *CCSDK_home***lib** directory.

   > **Note:**
   >
   > If you generate both a **ccCustom.jar** file using the Customer Center SDK and a **CustomerCenter**_*Java_locale*.**jar** file using the Localization SDK, you *must* use the same certificate to sign both JAR files.

10. Create a new folder called **custom** under the directory where you installed Customer Center on your Web server, *Customer_Center_home*.
11. Copy the **ccCustom.jar** file from *CCSDK_home***lib** to the *Customer_Center_home***custom** folder.
12. Deploy your custom localized JAR files.

    See "Deploying an Advanced Customization of Customer Center".

Deploying an Advanced Customization of Customer Center

If you are using a WebStart installation of Customer Center, use this procedure to deploy your custom localized JAR files, such as **ccCustom.jar**, **CustomerCenter**_*Java_locale*.**jar**, or both.

1. Go to the directory where you installed Customer Center on your Web server (your WebStart installation directory), *Customer_Center_home*.
2. Create a new folder named **custom** in *Customer_Center_home*.
3. Copy any modified JAR files to *Customer_Center_home***custom**.
4. Copy the **custom.jnlp** file from the *CCSDK_home***CustCntr\custom** folder to the *Customer_Center_home***custom** folder.
5. Uncomment the following line in the *Customer_Center_home***CustomerCenter**_*Java_locale*.**jnlp** file by removing the **<!--** and **-->** characters shown here:

   ```
   <!-- <extension name="Customized" href="custom/custom.jnlp"/> -->
   ```

   If you used the Localization SDK to create a custom version of **CustomerCenter**_*Java_locale*.**jar**, do the following also:

   a. Add this line:

      <jar href="custom/CustomerCenter_*Java_locale***.jar"/>**

      To the file *Customer_Center_home***custom\custom.jnlp** as shown here:

      ```
      <resources>
           <j2se version="1.4*"/>
           <jar href="custom/ccCustom.jar"/>
           <jar href="custom/CustomerCenter_Java_locale.jar"/>
      ```

   b. Remove the reference to the original **CustomerCenter**_*Java_locale*.**jar** file in *Customer_Center_home***CustomerCenter**_*Java_locale*.**jnlp** by making it an HTML comment:

      ```
      <!-- <jar href="lib/CustomerCenter_Java_locale.jar"/> -->
      ```

6. Restart Customer Center.

When to Use the Localization SDK for Advanced Customization

You might need to use the Localization SDK for these reasons:

- A few property files are not available in the Customer Center SDK; they are available only in the Localization SDK. If you need to change strings in these property files, you need the Localization SDK.
- Because the Customer Center SDK is in English, you may need to use the Localization SDK to identify the key name for the string you want to modify.

Finding key-value pairs to customize in the properties files

Several resource files are used to build the **CustomerCenter**_*Java_locale*.**jar** file.

When customizing Customer Center, you may not be able to find a key-value pair you need to modify for your localization in the properties files. To find keys for text in the user interface or for error messages, browse the following resource files:

- These resource files contain text that appears in the Customer Center UI:
  - **com/portal/app/comp/AppViewResources_1.properties**
  - **com/portal/app/cc/CustomerCenterResources_1.properties**
  - **com/portal/app/cc/comp/CCViewResources_1.properties**
- These resource files contain error messages that may appear in Customer Center error dialogs:
  - **com/portal/bas/comp/IAViewResources_1.properties**
  - **com/portal/bas/IACoreResources_1.properties**
- You can also find key-value pairs for the Event Browser in these resource files:
  - **com/portal/browse/BrowserResources_1.properties**
  - **com/portal/browse/EventTemplates_1.properties**
  - **com/portal/search/SearchResources_1.properties**

Localizing Self-Care Manager

To run your localized version of Self-Care Manager, you need a Web server and a servlet engine. For more information, see *BRM Installation Guide*.

To localize Self-Care Manager:

1. Verify that you have the W drive mapped to **C:\Program Files\Portal Software\Localization SDK**.
2. Download the English version of Self-Care Manager. Save the zip file to any temporary directory.
3. Install English Self-Care Manager. You do not need to install a Web server or servlet engine at this point. You need to install Self-Care Manager just to get access to its files, not to actually run it.

   For information, see installing BRM client applications in *BRM Installation Guide*.
4. Copy the file **webkit_en.war** from *Self-Care_Manager_install_dir***WebKit** to **W:***SDK_locale***zips**.

   If you installed Self-Care Manager in the default location, *Self-Care_Manager_install_dir* is **C:\Program Files\Portal Software\WebKit**.

   > **Note:**
   >
   > Create the **zips** directory if it does not already exist. The directory is not created by default.
5. Copy the Self-Care Manager zip file you downloaded from the temporary directory to **W:***SDK_locale***zips**.
6. Localize the file **web.xml** in **W:***SDK_locale***JAVA_PROJECTS\WebKit**, as follows:

a. Change each instance of **htmlui_en** to **htmlui**_*Java_locale*.

b. In this line:

```
<display-name>WebKit</display-name>
```

Change **WebKit** to something else, to distinguish it from the English version.

c. Save the file as **web**_*Java_locale*.**xml** in the same directory.

7. Go to **W:***SDK_locale***JAVA_PROJECTS\WebKit\webkit_ui** and take these steps for the HTML version of Self-Care Manager:

a. Copy **webkit_L10N_en.txt** to **webkit_L10N**_*Java_locale*.**txt**. For *Java_locale*, use the Java locale. For a list, see "Locale Names".

For example, if you are localizing into French, copy this file to **webkit_L10N_fr.txt**.

b. Translate the strings in **webkit_L10N**_*Java_locale*.**txt**. This file contains the strings that need to be localized in Self-Care Manager files. For more information, see "Translating the Self-Care Manager Localized Strings File".

8. Go to the **JAVA_PROJECTS** directory.

9. Set up the **CLIENTS_DEST** environment variable if you have not already set it:

```
set CLIENTS_DEST=W:\Client_dest
```

10. Enter this command to make the Self-Care Manager JAR file:

```
W:\SDK_locale\JAVA_PROJECTS> makeResourceJars locale webkit
```

11. Go to **W:***SDK_locale***bin**.

12. Enter this command to package the localized version of Self-Care Manager:

```
W:\SDK_locale\bin> Package_WebKit.bat locale
```

The localized Self-Care Manager package is created in the **W:***SDK_locale***zips** directory.

13. To install and run the localized version of Self-Care Manager, follow the installation instructions in installing Self-Care Manager in *BRM Installation Guide*.

Translating the Self-Care Manager Localized Strings File

Notes about localizing the **webkit_L10N_en.txt** file:

- This file should contain only UTF8 characters.
- The first set of variables are global variables that will be changed in each localized file. All localized files are listed in this file.
- Some file names are followed by a set of variables specific to that file. Other file names have no variables listed. Those files use only the global variables.
- Precede each line of comments with a # character.

Creating a Localized Self-Care Manager Installation for Linux

After you have completed localizing Self-Care Manager for Windows, you can package Self-Care Manager to run on Linux.

> **Note:**
>
> These steps do not apply to Windows.

1. On the Linux computer, create a directory for the English installation of Self-Care Manager.

   ```
   system% mkdir tarfile_dir
   ```

2. Download the Self-Care Manager Linux English version.
3. Go to the directory:

   ```
   system% cd tarfile_dir
   ```

4. Extract the Self-Care Manager **tar** file. For example:

   ```
   system% tar xvf SelfCareMgr_linux.tar
   ```

5. On your Windows system, unzip the localized **7.3.1_SelfCareMgr**_*Java_locale*.zip file you created for Windows to get the **webkit**_*Java_locale*.**war** file.
6. Use **ftp** to transfer **webkit**_*Java_locale*.**war** from your Windows system to your Linux system. Use binary transfer mode:

   ```
   ftp webkit_Java_locale.war Self-Care_Manager_dir/fg
   ```

7. Go to *Self-Care_Manager_dir* on the Linux system. This is the directory where you extracted the English **tar** file for the Linux version of Self-Care Manager.
8. Remove the English Self-Care Manager WAR file:

   ```
   system% rm Self-Care_Manager_dir/fg/WebKit/webkit_en.war
   ```

9. Make a **tar** file from the directory:

   ```
   system% tar cvf WebKit_Java_locale.tar Self-Care_Manager_dir
   ```

Localizing and Customizing Strings

Table 56-2 lists the files that you can customize to support your business model. For example, you can add new reasons to the **reasons.***locale* file or new payment channels to the **payment_channel.***locale* file. These files are all located in *BRM_home***/sys/msgs**.

Table 56-2 Files Associated with Localization and String Customization

File in *BRM_home*/sys/msgs/	Contains
active_mediation/active_mediation.en_US	List of subscriber preferences used for active mediation.
businessprofiles/business_profile_descr.en_US	List of business profile descriptions to display in a third-party CRM application.
devicestates/device_states.en_US	List of names you use for device states in the Device Management framework.
eradescr/era_descr.en_US	Descriptions of promotions for GSM Customer Center Extension.
errorcodes/errors.en_US	Server error messages.
featuresandprofilestates/features_and_profiles_states.en_US	Service provisioning states for GSM Customer Center Extension.

Table 56-2 (Cont.) Files Associated with Localization and String Customization

File in *BRM_home*/sys/msgs/	Contains
lifecycle_states/lifecycle_states.en_US	List of life cycle states used for subscriber life cycle management.
localedescr/locale_descr.en_US	Descriptions of the locales that BRM supports, used by Customer Center.
newsfeed/newsfeed.en_US	List of localized strings for News Feed.
note/note.en_US	List of types and permissible statuses of a **/note** object.
numcategories/num_categories.en_US	Number attributes for Number Administration Center.
numdevicestates/num_device_states.en_US	List of status settings for a number in Number Administration Center and Customer Center.
numvanities/num_vanities.en_US	List of vanity numbers in Number Administration Center.
ordersimstatus/order_sim_status.en_US	List of status settings for an order in SIM Administration Center.
paymentchannel/payment_channel.en_US	List of payment channel IDs that can be included in payments received by BRM.
reasoncodes/reasons.en_US	List of reasons for account changes (such as charges and credits) in Customer Center.
revenueassurance/ra_alert_message.en_US	List of alert messages that BRM uses when notifying analysts that a revenue assurance threshold has been exceeded.
simcardtypes/sim_card_types.en_US	List of SIM card types for SIM Administration Center.
simdevicestates/sim_device_states.en_US	List of status settings for SIM devices in SIM Administration Center.
suspense_reason_code/suspense_reason_code.en_US	List of reasons for call failures in Suspense Management Center.
system_filter_set/system_filterset_edr_field_values.en_US	List of EDR fields and values that Pipeline Manager uses as filtering criteria for system products and discounts.
telcofeaturesandprofilestates/ telco_features_and_profiles_states.en_US	Service provisioning states for GSM Customer Center Extension. These states are specific to telco.
voucher_devicestates/device_state_voucher.en_US	List of status settings for voucher devices in Voucher Administration Center.
voucher_orderstates/order_state_voucher.en_US	List of order statuses to display in Voucher Administration Center.

Creating New Strings and Customizing Existing Strings

To create new strings or customize existing strings:

1. Open the appropriate file.
2. Locate and edit any existing strings you want to customize.
3. Add any new strings using the standard format.
4. Save the file.

5. Load the strings using the **load_localized_strings** utility. See "Loading Localized or Customized Strings".

> **Note:**
>
> Use the appropriate file suffix for the locale. For example, to load SIM card formats in the German language, use the file named **sim_card_types.de**. See "Locale Names".

Localizing Existing Strings

> **Note:**
>
> If you are only localizing the strings:
>
> - *Do not* edit the ID, DOMAIN, or VERSION strings in these files.
> - Be sure to preserve the double quotes (") and semicolon (;) around any strings that you change.
> - There must be a space before a semicolon (;).

To localize any of these files:

1. Make a copy of the file and open it.
2. Change the LOCALE to the BRM name for the new locale.

   For BRM locale names, see "Locale Names".
3. Translate the text strings for STRING and HELPSTR.
4. Rename the file by changing the suffix from **.en_US** to the BRM locale name for your locale. For example, a Traditional Chinese version of **errors.en_US** is named **errors.zh_TW**.
5. Convert the file to UTF8 encoding.

Loading Localized or Customized Strings

You use the **load_localized_strings** utility to load the contents of a string file into a **/strings** object in the BRM database.

> **Caution:**
>
> When loading reason codes from the **reasons.***locale* file, **load_localized_strings** also loads information from this file into the **/config/map_glid** object. If customized to specify service types and event types for event-level adjustments, the utility also loads information into the **/config/reason_code_scope** object. Though the utility does not overwrite existing strings in the **/strings** object unless you direct it to, it *does* overwrite the **/config/reason_code_scope** and **/config/map_glid** objects.

> **Note:**
>
> - The **load_localized_strings** utility needs a configuration file in the directory from which you run the utility. See the discussion about creating configuration files for BRM utilities in *BRM System Administrator's Guide*.

1. Use the following command to run the **load_localized_strings** utility:

   ```
   load_localized_strings string_file_name.locale
   ```

   For example:

   ```
   load_localized_strings sim_device_states.en_US
   ```

2. Look in the **load_localized_strings.log** file to find any errors. The log file is either in the directory from which the utility was started or in a directory specified in the configuration file.
3. Verify that the strings were loaded by displaying the **/strings** objects using the Object Browser or the **robj** command with the **testnap** utility. See "Reading an Object and Writing Its Contents to a File" for information on how to use Object Browser. See "Using the testnap Utility to Test BRM" for general instructions on using **testnap**.

   > **Note:**
   >
   > For the **reasons.***locale* file, you should also use one of these methods to check the **/config/map_glid** and **/config/reason_code_scope** objects.

4. Stop and restart the Connection Manager (CM). For more information, see Starting and Stopping the BRM System in *BRM System Administrator's Guide*.
5. If the strings are displayed in a GUI application, stop and restart the application to display the strings.

Localizing BRM Reports

To create localized versions of BRM report templates, use their conversion routines. For more information, see "Localizing BRM Reports".

About Customizing Server Software

This section provides an introduction to customizing server software.

Setting the Default Language for Customer Accounts

The **/account** object contains the PIN_FLD_LOCALE field. During BRM account creation, the CSR can set the value of this field using the BRM locale list in Customer Center. By default in Customer Center, the PIN_FLD_LOCALE field is set to the locale of the CSR's system.

The PIN_FLD_LOCALE field is set by the PCM_OP_CUST_SET_LOCALE opcode. This opcode reads the PCM_OP_CUST_POL_PREP_LOCALE and

PCM_OP_CUST_POL_VALID_LOCALE policy opcodes. When you develop a Web page for creating customer accounts, you need to set this field.

Customizing Canonicalization

To customize canonicalization, use the PCM_OP_CUST_POL_CANONICALIZE opcode.

This opcode is called by PCM_OP_CUST_COMMIT_CUSTOMER, PCM_OP_CUST_SET_NAMEINFO, and the Customer Center search screen. PCM_OP_CUST_POL_CANONICALIZE searches for localized (non-English) customer input string fields.

The default implementation of the PCM_OP_CUST_POL_CANONICALIZE opcode is the en_US locale. Canonicalization handles Latin-based characters only. You must customize this opcode for other languages.

Exporting Data to an LDAP Server

For languages other than English, the data exported to the directory structure using Lightweight Directory Access Protocol (LDAP) is in UTF8 format. If you need native encodings, you must write conversion applications that operate on the data in the directory server to convert it in place.

Locale Names

Following is a list of locales for which BRM supports canonicalization. You can also use this list to get common BRM, Linux, and Java locale names. For information on canonicalization, see "About Localizing and Internationalizing".

In naming localized files, use the more general two-letter locale names when possible, but in some cases the more specific four-letter locale names are needed. For example:

- Use **zh_TW** for Traditional Chinese.
- Use **en_US** for English files.
- Use **pt_BR** for Brazilian Portuguese.

Table 56-3 lists many common locale names. You can get complete locale name lists on the Web.

Table 56-3 Common Locale Names

Language (ISO-639)	Country (ISO-3166)	Linux, Java, and BRM Locale	Windows Locale
Chinese (Simplified)	People's Republic of China	zh_CN	CHS
Chinese (Traditional)	Republic of China (Taiwan)	zh_TW	CHT
Danish	Denmark	da_DK	DAN
Dutch	Netherlands	nl_NL	NLD
Dutch	Belgium	nl_BE	NLB
English	Australia	en_AU	ENA
English	Canada	en_CA	ENC
English	Ireland	en_IE	ENI
English	New Zealand	en_NZ	ENZ

Table 56-3 (Cont.) Common Locale Names

Language (ISO-639)	Country (ISO-3166)	Linux, Java, and BRM Locale	Windows Locale
English	South Africa	en_ZA	ENS
English	United Kingdom	en_GB (BRM, Java and Linux)	ENG
English	United States	en_US	ENU
Finnish	Finland	fi_FI	FIN
French	France	fr _FR	FRA
French	Belgium	fr_BE	FRB
French	Canada	fr_CA	FRC
French	Luxembourg	fr_LU	FRL
French	Switzerland	fr_CH	FRS
German	Germany	de_DE	DEU
German	Austria	de_AT	DEA
German	Luxembourg	de_LU	DEL
German	Switzerland	de_CH	DES
Japanese	Japan	ja_JP	JPN
Korean	Korea	ko_KR	KOR
Italian	Italy	it_IT	ITA
Italian	Switzerland	it_CH	ITS
Norwegian (Bokmal)	Norway	no_NO	NOR
Norwegian (Nynorsk)	Norway	no_NY (BRM and Linux) no_NO_NY (Java)	NON
Portuguese	Portugal	pt_PT	PTG
Portuguese	Brazil	pt_BR	PTB
Spanish	Spain	es_ES	ESP
Spanish	Argentina	es_AR	ESS
Spanish	Bolivia	es_BO	ESB
Spanish	Chile	es_CL	ESL
Spanish	Colombia	es_CO	ESO
Spanish	Costa Rica	es_CR	ESC
Spanish	Dominican Republic	es_DO	ESD
Spanish	Ecuador	es_EC	ESF
Spanish	El Salvador	es_SV	ESE
Spanish	Guatemala	es_GT	ESG
Spanish	Mexico	es_MX	ESM
Spanish	Nicaragua	es_NI	ESI
Spanish	Panama	es_PA	ESA
Spanish	Paraguay	es_PY	ESZ
Spanish	Peru	es_PE	ESR
Spanish	Puerto Rico	es_PR	ESU

Table 56-3 (Cont.) Common Locale Names

Language (ISO-639)	Country (ISO-3166)	Linux, Java, and BRM Locale	Windows Locale
Spanish	Uruguay	es_UY	ESY
Spanish	Venezuela	es_VE	ESV
Swedish	Sweden	sv_SE	SVE

57

Handling Non-ASCII Code on the BRM Server

Learn how to use character-encoding conversion-layer macros with languages other than English (that is, any non-ASCII character encoding) in your Oracle Communications Billing and Revenue Management (BRM) system.

Topics in this document:

- About Character-Encoding Conversion
- About Converting Multibyte or Unicode to and from UTF8
- Direct Conversion Macros
- Supporting Functions and Macros
- Universal Macros
- Conversion Code Example

About Character-Encoding Conversion

To work with different languages, BRM applications must use a character encoding that supports them. To support any Western European language or East Asian language, the macros described in this document are required. You must use the conversion macros with any BRM client, server, or Web application localization that is *not* written in Java. Without these macros, only the 7-bit ASCII encoding works.

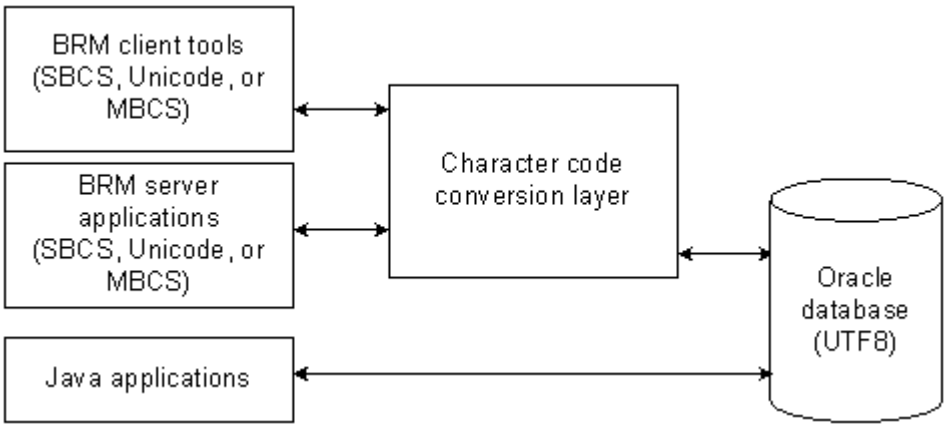
> **Note:**
>
> These macros are not required for English language applications, but using the macros facilitates later translations.

BRM supports localizations using Latin 1 and some of the East Asian encodings for Japanese, Korean, Traditional Chinese, and Simplified Chinese only.

About Converting Multibyte or Unicode to and from UTF8

Figure 57-1 shows the relationships between the BRM applications and the character-encoding conversion layer:

Figure 57-1 BRM Applications and the Character-Encoding Conversion Layer



If you are developing a command-line application or a third-party integration, use the macros that convert either Unicode or multibyte input strings to UTF8 and from UTF8:

- PIN_CONVERT_STR_TO_UTF8
- PIN_CONVERT_UTF8_TO_STR

The macros check the defined preprocessor directive (**_MBCS** and **_UNICODE**) to call the direct conversion macros and call the supporting function and macro.

If you are developing a BRM client application or working with multibyte or Unicode only, use the direct conversion macros to change the character encoding. See "About Converting Multibyte or Unicode to and from UTF8":

- PIN_CONVERT_MBCS_TO_UTF8
- PIN_CONVERT_UTF8_TO_UNICODE
- PIN_CONVERT_UNICODE_TO_UTF8
- PIN_CONVERT_UTF8_TO_MBCS

The macros are located in the Portal Communication Module (PCM) library. The header file is in **pcm.h**.

Direct Conversion Macros

Table 57-1 lists the functions available in the direct conversion macros.

Table 57-1 Direct Conversion Macros

Function	Description
PIN_CONVERT_MBCS_TO_UTF8	Converts a multibyte character string to UTF8.
PIN_CONVERT_UNICODE_TO_UTF8	Converts Unicode characters to UTF8.
PIN_CONVERT_UTF8_TO_MBCS	Converts UTF8 characters to multibyte.
PIN_CONVERT_UTF8_TO_UNICODE	Converts a UTF8 character string to a Unicode string.

Supporting Functions and Macros

Table 57-2 lists the supporting functions and macros.

Table 57-2 Supporting Functions and Macros

Function	Description
pin_IsValidUtf8	Determines whether a specified string is using UTF8 encoding.
PIN_MBSLEN	Determines the length of the multibyte string.
PIN_SETLOCALE	Sets, changes, or queries some or all of the current program locale, specified by locale and category.

Universal Macros

Table 57-3 lists the universal macros.

Table 57-3 Universal Macros

Function	Description
PIN_CONVERT_STR_TO_UTF8	Converts translatable database data to UTF8.
PIN_CONVERT_UTF8_TO_STR	Calls PIN_CONVERT_UTF8_TO_MBCS when **_MBCS** is defined, and calls PIN_CONVERT_UTF8_TO_UNICODE when **_UNICODE** is defined. This macro is called whenever translatable data is retrieved from the database.

PIN_CONVERT_MBCS_TO_UTF8

This macro converts a multibyte character string to UTF8.

> **Note:**
>
> You need to use **setlocale** before calling this macro.

Syntax

```
int32
PIN_CONVERT_MBCS_TO_UTF8(
                char            *pLocaleStr,
                char            *pMultiByteStr,
                int32            nMultiByteLen,
                unsigned char   *pUTF8Str,
                int32            nUTF8size,
                pin_errbuf_t    *ebufp);
```

Parameters

pLocaleStr
Indicates the locale of the multibyte string input. The locale string argument can have following values:

- **en_US** - US-English locale
- **""** - System default locale
- **NULL** - Where LC_CTYPE is set to the appropriate locale before calling the macro

pMultiByteStr
Points to the character string to be converted.

nMultiByteLen
Specifies the number of characters to be converted in the string pointed to by *pMultiByteStr*. If this value is **1**, the string is assumed to be NULL-terminated and the length is calculated automatically.

pUTF8Str
Points to the buffer that receives the converted UTF8 string.

nUTF8size
Specifies the size of the buffer pointed to by *pUTF8Str*.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.

Return Values

Table 57-4 lists the values returned by PIN_CONVERT_MBCS_TO_UTF8.

Table 57-4 PIN_CONVERT_MBCS_TO_UTF8 Return Values

SourceString pMultiByteStr	Number of Characters to Convert in Input String nMultiByteLen	Buffer pUTF8Str	Buffer Size in Bytes nUTF8size	Return Value
Null terminated	Any number	pMultiByteStr = pUTF8Str	Any size	0 (ERR)
NULL	Any number	Any	Any size	0 (ERR)
Any	< -1	Any	Any size	0 (ERR)
Any	0	Any	Any size	0 (ERR)
Null terminated	-1 or > 0	NULL	!=0	0 (ERR)
Not null terminated	>0	NULL	!=0	0 (ERR)
Any	Any number	!=NULL	<=0	0 (ERR)
Not null terminated	Any number	pMultiByteStr = pUTF8Str	Any size	0 (ERR)
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Not null terminated	> 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	NULL	0	Required buffer size

Table 57-4 (Cont.) PIN_CONVERT_MBCS_TO_UTF8 Return Values

SourceString pMultiByteStr	Number of Characters to Convert in Input String nMultiByteLen	Buffer pUTF8Str	Buffer Size in Bytes nUTF8size	Return Value
Not null terminated	-1 or > 0	NULL	0	Required buffer size
Null terminated	-1 or > 0	NULL	0	Required buffer size

Error Handling

If *pMultiByteStr* and *pUTF8Str* are the same, the macro fails and the error buffer returns **PIN_ERR_BAD_ARG**. If the macro encounters an invalid character in the source string, the macro fails; it sets the return value to **0** and the error buffer to the respective error code as shown in Table 57-5:

Table 57-5 Error Codes

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_BAD_ARG	4	0	Bad argument
PIN_ERR_BAD_LOCALE	71	0	Invalid locale string
PIN_ERR_CONV_MULTIBYTE	72	0	Error in multibyte to UTF8 conversion
PIN_ERR_NULL_PTR	39	0	Empty string passed

PIN_CONVERT_STR_TO_UTF8

This macro converts translatable database data to UTF8.

Syntax

```
int32
PIN_CONVERT_STR_TO_UTF8(
                char              *pLocaleStr,
                char              *pSourceStr,
                int32              nSourceLen,
                unsigned char     *pUTF8Str,
                int32              nUTF8size,
                pin_errbuf_t      *ebufp);
```

PIN_CONVERT_UNICODE_TO_UTF8

This macro converts Unicode characters to UTF8.

> **Note:**
>
> You need to use **setlocale** before calling this macro.

Syntax

```
int32
PIN_CONVERT_UNICODE_TO_UTF8(
                wchar_t          *pUnicodeStr,
                int               nUnicodeLen,
                unsigned char    *pUTF8Str,
                int               nUTF8,
                pin_errbuf_t     *ebufp);
```

Parameters

pUnicodeStr
Points to the character string to be converted.

nUnicode
Specifies the number of characters to be translated in the string pointed to by *pUnicodeStr*.

pUTF8Str
Points to a buffer that receives the translated UTF8 string.

nUTF8
Specifies the size of the buffer pointed to by *pUTF8Str*.

Return Values

Returns **NULL** in the error buffer if the macro is successful. Returns the cause of the error if the macro fails.

Error Handling

If *pUnicodeStr* and *pUTF8Str* are the same, the macro fails, and the error buffer returns **PIN_ERR_BAD_ARG**. If the macro encounters an invalid character in the source string, the macro fails; it sets *nUTF8* to **0** and sets the error buffer to the respective error code as shown in Table 57-6:

Table 57-6 Error Handling Codes

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_BAD_ARG	4	0	Bad argument
PIN_ERR_CONV_UNICODE	73	1	Error in UTF8 to Unicode conversion
PIN_ERR_BAD_UTF8	75	0	Invalid UTF8 characters
PIN_ERR_NULL_PTR	39	0	Empty string passed

PIN_CONVERT_UTF8_TO_MBCS

This macro converts UTF8 characters to multibyte.

> **Note:**
>
> You need to use **setlocale** before calling this macro.

Syntax

```
int32
PIN_CONVERT_UTF8_TO_MBCS(
                char              *pLocaleStr,
                unsigned char     *pUTF8Str,
                int                nUTF8Len,
                char              *pMultiByteStr,
                int                nMultiByte,
                pin_errbuf_t      *ebufp);
```

Parameters

pLocaleStr
Indicates the locale of the multibyte string input. The locale string argument can have following values:

- **en_US** - US-English locale
- **""** - System default locale
- **NULL** - Where LC_CTYPE is set to the appropriate locale before calling the macro

pUTF8Str
Points to the character string to be converted.

nUTF8Len
Specifies the number of bytes to be converted in the string pointed to by *pUTF8Str*.

pMultiByteStr
Points to the buffer that receives the converted multibyte string.

nMultiByte
Specifies the size of the buffer pointed to by *pMultiByteStr*.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.

Return Values

Table 57-7 lists the values returned by PIN_CONVERT_UTF8_TO_MBCS.

Table 57-7 Values Returned by PIN_CONVERT_UTF8_TO_MBCS

Source String pUTF8Str	Number of Characters to be Converted in Input String nUTF8Len	Buffer pMultibyteStr	Buffer Size in Bytes nMultibyte	Returned Value
Null terminated	Any number	pMultibyteStr = pUTF8Str	Any size	0 (ERR)
NULL	Any number	Any	Any size	0 (ERR)
Any	< -1	Any	Any size	0 (ERR)
Any	0	Any	Any size	0 (ERR)
Null terminated	-1 or > 0	NULL	!=0	0 (ERR)
Not null terminated	>0	NULL	!=0	0 (ERR)

Table 57-7 (Cont.) Values Returned by PIN_CONVERT_UTF8_TO_MBCS

Source String pUTF8Str	Number of Characters to be Converted in Input String nUTF8Len	Buffer pMultibyteStr	Buffer Size in Bytes nMultibyte	Returned Value
Any	Any number	!=NULL	<=0	0 (ERR)
Not null terminated	Any number	pMultibyteStr = pUTF8Str	Any size	0 (ERR)
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Not null terminated	> 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	NULL	0	Required buffer size
Not null terminated	-1 or > 0	NULL	0	Required buffer size
Null terminated	-1 or > 0	NULL	0	Required buffer size

Error Handling

If *pMultiByteStr* and *pUTF8Str* are the same, the macro fails and the error buffer returns **PIN_ERR_BAD_ARG**. If the macro encounters an invalid character in the source string, the macro fails; it sets the return value to **0** and the error buffer to **PIN_ERR_BAD_UTF8** as shown in Table 57-8:

Table 57-8 Error Handling Codes

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_BAD_ARG	4	0	Bad argument
PIN_ERR_BAD_LOCALE	71	0	Invalid locale string
PIN_ERR_CONV_MULTIBYTE	72	1	Error in UTF8 to multibyte conversion
PIN_ERR_BAD_UTF8	75	0	Invalid UTF8 characters
PIN_ERR_NULL_PTR	39	0	Empty string passed
PIN_ERR_NO_MEM	1	1	Can't allocate enough memory for conversion

PIN_CONVERT_UTF8_TO_STR

This macro calls "PIN_CONVERT_UTF8_TO_MBCS " when _**MBCS** is defined and calls "PIN_CONVERT_UTF8_TO_UNICODE " when _**UNICODE** is defined. This macro is called whenever translatable data is retrieved from the database.

Syntax

```
int32
PIN_CONVERT_UTF8_TO_STR(
                char            *pLocaleStr,
                char            *pUTF8Str,
                int32            nUTF8Len,
                unsigned char   *pBuffer,
```

```
                int32              nBuffersize,
                pin_errbuf_t      *ebufp);
```

PIN_CONVERT_UTF8_TO_UNICODE

This macro converts a UTF8 character string to a Unicode string.

Syntax

> **Note:**
>
> You need to use **setlocale** before calling this macro.

```
int32
PIN_CONVERT_UTF8_TO_UNICODE(
                unsigned char     *pUTF8Str,
                int                nUTF8,
                wchar_t           *pUnicodeStr,
                int                nUnicode,
                pin_errbuf_t      *ebufp);
```

Parameters

pUTF8Str
Points to the character string to be converted.

nUTF8
Specifies the number of bytes to be converted in the string pointed to by *pUTF8Str*.

pUnicodeStr
Points to a buffer that receives the converted Unicode string.

nUnicode
Specifies the size of the buffer pointed to by *pUnicodeStr*.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.

Return Values

Table 57-9 lists the return values for PIN_CONVERT_UTF8_TO_UNICODE.

Table 57-9 Values Returned by PIN_CONVERT_UTF8_TO_UNICODE

Source String pUTF8str	Number of Bytes to Convert in Input String nUTF8	Buffer pUnicodeSrt	Buffer Size in Bytes nUnicode	Returned Value
Null terminated	Any number	pUTF8str = pUnicodeSrt	Any size	0 (ERR)
NULL	Any number	Any	Any size	0 (ERR)
Any	< -1	Any	Any size	0 (ERR)
Any	0	Any	Any size	0 (ERR)

Table 57-9 (Cont.) Values Returned by PIN_CONVERT_UTF8_TO_UNICODE

Source String pUTF8str	Number of Bytes to Convert in Input String nUTF8	Buffer pUnicodeSrt	Buffer Size in Bytes nUnicode	Returned Value
Null terminated	-1 or > 0	NULL	!=0	0 (ERR)
Not null terminated	>0	NULL	!=0	0 (ERR)
Any	Any number	!=NULL	<=0	0 (ERR)
Not null terminated	Any number	pUTF8str = pUnicodeSrt	Any size	0 (ERR)
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Not null terminated	> 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	!=NULL	>0	Converted characters
Null terminated	-1 or > 0	NULL	0	Required buffer size
Not null terminated	-1 or > 0	NULL	0	Required buffer size
Null terminated	-1 or > 0	NULL	0	Required buffer size

Error Handling

If *pUnicodeStr* and *pUTF8Str* are the same, the macro fails and the error buffer returns **PIN_ERR_BAD_ARG**. If the macro encounters an invalid character in the source string, the macro fails; it sets the return value to **0** and the error buffer to **PIN_ERR_BAD_UTF8** as shown in Table 57-10:

Table 57-10 Error Handling Codes

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_BAD_ARG	4	0	Bad argument
PIN_ERR_CONV_UNICODE	73	1	Error in UTF8 to Unicode conversion
PIN_ERR_BAD_UTF8	75	0	Invalid UTF8 characters

pin_IsValidUtf8

This function determines whether a specified string is using UTF8 encoding.

Syntax

```
int32
pin_IsValidUTF8(
                unsigned char     *pUTF8Str,
                int32              nUTF8Len,
                pin_errbuf_t      *ebufp);
```

Parameters

pUTF8Str
Points to the character string to be checked for UTF8 encoding.

nUTF8Len
Specifies the number of bytes to be checked in the string pointed to by *pUTF8Str*.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.

Return Values

Returns a positive value if *pUTF8Str* is a valid UTF8 string.

Error Handling

This macro returns **0** if *pUTF8Str* is not a valid UTF8 string or if any errors occur. Table 57-11 lists the error codes.

Table 57-11 Error Handling Codes

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_NULL_PTR	39	0	Empty string passed
PIN_ERR_BAD_UTF8	75	0	Invalid UTF8 characters

PIN_MBSLEN

This macro determines the length of the multibyte string.

Syntax

```
int32
PIN_MBSLEN(
                char            *pLocaleStr,
                char            *pMultiByteStr,
                pin_errbuf_t    *ebufp);
```

Parameters

pLocaleStr
Indicates the locale information of the input multibyte string. The locale string argument can have the following values:

- **en_US** - US-English locale
- **""** - System default locale
- **NULL** - Where LC_CTYPE is set to the appropriate locale before calling the macro

pMultiByteStr
Points to the multibyte character string.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.

Return Values

Returns the length of the multibyte string.

Error Handling

This macro returns **0** if any errors occur.

PIN_SETLOCALE

This macro sets, changes, or queries some or all of the current program locale, specified by locale and category. Locale-dependent categories include date and currency formats.

Syntax

```
char*
PIN_SETLOCALE(
               const int         n_category,
               char             *locale_p,
               pin_errbuf_t     *ebufp);

*int32
PIN_MBSLEN(
               const int         ncategory,
               char             *locale_p,
               pin_errbuf_t     *ebufp);
```

Parameters

n_category
The parts of a program's locale that are affected. The macros used for category and the parts of the program they affect are:

- LC_ALL – All categories, as listed below.
- LC_COLLATE – The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, and **strxfrm** macros.
- LC_CTYPE – The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).
- LC_MONETARY – Monetary format information returned by the **localeconv** function.
- LC_NUMERIC – Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the noncurrency formatting information returned by **localeconv**.
- LC_TIME – The **strftime** and **wcsftime** functions.

locale_p
Indicates the locale.

ebufp
A pointer to the error buffer. If this macro is successful, the error buffer is **NULL**; otherwise, it indicates the cause of the error.
The null pointer is a special directive that tells PIN_SETLOCALE to query rather than set the international environment.

Return Values

Returns a pointer to the string associated with the specified locale and category.

Error Handling

If the locale or category is invalid, the macro returns a null pointer and sets the error buffer to PIN_ERR_BAD_LOCALE as shown in Table 57-12:

Table 57-12 Error Handling Code

Returned Error Code	Value	Reserved Bit in ebuf	Error Description
PIN_ERR_BAD_LOCALE	71	0	Invalid locale string

Conversion Code Example

The following code sample shows how to get the locale from the account information and convert the locale from BRM locale to platform locale:

```
vp = PIN_FLIST_FLD_GET(tmp_acctinfo_flistp, PIN_FLD_LOCALE,
            1, ebufp);
     if (vp == NULL) { /* Set default locale */
            strcpy(infranet_locale, "en_US");
    } else {
            strcpy(infranet_locale, (char *)vp);
    }
    locale = PIN_MAP_INFRANET_TO_PLATFORM_LOCALE(infranet_locale,
            ebufp);
```

The following function shows how to convert a UTF8 string to an MBCS string:

```
static char *
fm_inv_pol_convert_utf8_to_str(
        char                    *orig_str,
        char                    *locale,
        pin_errbuf_t            *ebufp)
{
        int             orig_size = 0;
        int             dest_size = 0;
        char        *strbuf = NULL;

        orig_size = strlen((char *)orig_str) + 1;

        /* First round, get the required buffer size for output string */
        dest_size = PIN_CONVERT_UTF8_TO_MBCS(locale,
        (unsigned char *)orig_str,
        orig_size, NULL, 0, ebufp);

    if (dest_size == 0) {
        if( PIN_ERR_IS_ERR( ebufp )) {
            PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_DEBUG,
                "PIN_CONVERT_UTF8_TO_MBCS Failed",
                ebufp);
            PIN_ERR_CLEAR_ERR(ebufp);
        }
        return NULL;
    }

    strbuf = (char *)pin_malloc(sizeof(char)*(dest_size + 1));
    if (strbuf == NULL) {
        pin_set_err(ebufp, PIN_ERRLOC_FM,
            PIN_ERRCLASS_SYSTEM_DETERMINATE,
            PIN_ERR_NO_MEM, 0, 0, 0);
        PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_DEBUG,
            "PIN_CONVERT_UTF8_TO_MBCS Failed",
            ebufp);
        PIN_ERR_CLEAR_ERR(ebufp);
```

```
        return NULL;
    }

    /* Second round, do the string conversion */
    dest_size = PIN_CONVERT_UTF8_TO_MBCS(locale,
        (unsigned char *)orig_str,
        orig_size, strbuf, dest_size + 1, ebufp);

    if (dest_size == 0) {
        if( PIN_ERR_IS_ERR( ebufp )) {
            PIN_ERR_LOG_EBUF(PIN_ERR_LEVEL_DEBUG,
                "PIN_CONVERT_UTF8_TO_MBCS Failed",
                ebufp);
            PIN_ERR_CLEAR_ERR(ebufp);
        }
        pin_free(strbuf);
        return NULL;
    }

    return strbuf;
}
```

Part VIII
Programming Utilities

This part provides reference information about Oracle Communications Billing and Revenue Management (BRM) developer utilities. It contains the following chapter:

- Developer Utilities

58

Developer Utilities

Learn about the Oracle Communications Billing and Revenue Management (BRM) developer utilities.

Topics in this document:

- load_config
- load_config_provisioning_tags
- load_localized_strings
- load_pin_config_business_type
- load_pin_device_permit_map
- load_pin_device_state
- load_pin_excluded_logins
- load_pin_order_state
- parse_custom_ops_fields
- pin_adu_validate
- pin_bus_params
- pin_cfg_bpdump
- pin_crypt_app
- pin_deploy
- pin_uei_deploy
- pin_virtual_time
- testnap
- pin_config_editor
- generate_config_xml_from_opcode_spec.py

load_config

Use this utility to load the contents of XML configuration files into configuration (**/config/***) objects in the BRM database.

> **Note:**
>
> To connect to the BRM database, this utility needs a configuration (**pin.conf**) file in the directory from which it is run. For information about creating configuration files for BRM utilities, see "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Before loading the contents of a file, the utility validates the contents against the file's XML schema definition (XSD). To do this, the utility needs the **pin.conf** entries in Table 58-1.

Table 58-1 load_config pin.conf Entries Required for XML Validation

XML File to Validate	Required pin.conf Entry
config_lifecycle_states.xml	- load_config validation_module libLoadValidSLM LoadValidSLM_init
config_service_state_map.xml	- load_config validation_module libLoadValidSLM LoadValidSLM_init
config_subscriber_preferences_map.xml	- load_config validation_module libLoadValidTCFAAA LoadValidTelcoAAA_init
config_collections_scenario_params.xml	- load_config validation_module libLoadValidCollections LoadValidCollections_init

The location of the XSD must be specified in the XML file. If the contents do not conform to the XSD, the load operation fails.

After validating the XML contents against the XSD, the utility converts the XML file into one or more configuration objects, depending on the structure of the XML file.

If any syntax errors occur, the load operation fails.

Location

BRM_home/**apps/load_config**

Syntax

```
load_config [-d] [-h] [-l] [-n] [-r class] [-t] [-T] [-v] [-w] [class] [config_file]
```

Parameters

-d
Creates a log file for debugging purposes. Use this parameter for debugging when the utility appears to have run with no errors but the configuration objects have not been loaded into the database.

-h
Displays syntax and parameters for the utility.

-l
Lists all validated configuration objects in the BRM database that are associated with *config_file*.

-n
Creates a configuration object when a validation library is not loaded. Use this parameter to skip the validation and to load the contents of the XML file into the database schema when the corresponding validation library is not loaded. When the library is loaded, validation happens irrespective of whether this parameter is passed or not.
In the absence of the validation library and this parameter, the utility returns an error and exits.

-r *class*
Deletes all instances of the specified storable class from the BRM database.
To specify the *class*, do not include **/config/**. For example, to specify the **/config/lifecycle_states** storable class, enter **lifecycle_states**.

-t
Runs the utility in test mode to validate the XML file against its XML schema definition. This parameter does not load data into a configuration object or overwrite any existing data in the objects.

> **Tip:**
>
> To avoid load errors based on XML content problems, run the utility with this option *before* loading data into the object.

-T
Validates configuration objects in the BRM database that are associated with *config_file*.

-v
Displays information about successful or failed processing as the utility runs.

-w *class*
Writes information from all objects of the specified storable class in the database to *config_file*. To specify the *class*, do not include **/config/**. For example, to specify the **/config/ lifecycle_states** storable class, enter **lifecycle_states**.
This parameter can be used to copy configuration data from one system to another. The generated XML does not conform to a particular XSD. Before loading the contents of *config_file* into another database schema, you must associate it with an XSD that defines the appropriate format.

config_file
The XML file for which the utility performs one of the following actions:

- Loads the file's contents into the database schema
- Loads information from objects in the database schema into the file

This must be the last parameter listed on the command line.
If the XML file is in the same directory from which you run the utility, specify only the file name. If it is in a different directory, include the entire path for the file.

Results

If the utility does not notify you that it was successful, look in the **default.pinlog** file to find any errors. This log file is either in the directory from which the utility was run or in a directory specified in the utility's configuration file.

To verify that the configuration information was loaded, display the configuration objects by using one of the following features:

- Object Browser
- **testnap** utility's **robj** command

See "Reading an Object and Writing Its Contents to a File" for more information.

> **Note:**
>
> You must stop and restart the Connection Manager (CM) to make new service life cycle and service state mapping data available.

load_config_provisioning_tags

Use the **load_config_provisioning_tags** utility to load provisioning tags into the **/config/provisioning_tag** object in the BRM database. You define provisioning tags in the **pin_config_provisioning_tags.xml** file in *BRM_home***/sys/data/config**.

Use this utility when you create provisioning tags using the provisioning tag framework.

For information about creating provisioning tags, see "Creating Provisioning Tags" in *BRM Provisioning Services*.

For information about the syntax of the XML file, see "Configuring Provisioning Tags" in *BRM Provisioning Services*.

> **Caution:**
>
> The **load_config_provisioning_tags** utility overwrites existing instances of the **/config/provisioning_tag** object. If you are updating provisioning tags, you cannot load new or changed tags only. You must load the complete set of provisioning tags each time you run the **load_config_provisioning_tags** utility.

When you run this utility, the **pin_config_provisioning_tags.xml** and **business_configuration.xsd** files must be in the same directory. By default, both files are in *BRM_home***/sys/data/config**.

> **Note:**
>
> To connect to the BRM database, the **load_config_provisioning_tags** utility needs a configuration file in the directory from which you run the utility. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

*BRM_home***/bin**

Syntax

```
load_config_provisioning_tags [-d] [-v] [-t] [-h] pin_config_provisioning_tags_file
```

Parameters

-d
Creates a log file for debugging purposes. Use this parameter for debugging when the utility appears to have run with no errors, but the data has not been loaded into the database.

-v
Displays information about successful or failed processing as the utility runs.
load_pin_config_provisioning_tags *any_other_parameter* **–v >** *filename***.log**

-t
Runs the utility in test mode to validate the XML file. This parameter does *not* create, modify, or delete any entries in the **/config/provisioning_tag** object.

> **Tip:**
>
> To avoid load errors based on XML content problems, run the utility with this option *before* loading data into the database.

-h
Displays help information for using this utility.

pin_config_provisioning_tags_file
The name and location of the file that defines provisioning tags. The default **pin_config_provisioning_tags.xml** file is in *BRM_home***/sys/data/config**. The utility can take any XML file name as a parameter if the file's contents conform to the appropriate schema definition.

> **Note:**
>
> The file must be in the same directory as the **business_configuration.xsd** file.

If you do not run the utility from the directory in which the file is located, you must include the complete path to the file, for example:

```
load_config_provisioning_tags BRM_home/sys/data/config/pin_config_provisioning_tags.xml
```

Validity Checks

The utility validates the XML file against rules defined in the **business_configuration.xsd** file. This file resides in the *BRM_home***/sys/data/config** directory.

The utility validates the following:

- The service name length is from 1 to 1,023 characters.
- The service names listed in the file are unique.

Results

If **load_config_provisioning_tags** doesn't notify you that it was successful, look in the log file (normally **default.pinlog**) for error messages. The log file is located in the directory from which the utility was started or in a directory specified in the **pin.conf** configuration file.

> **Note:**
>
> You must restart the Connection Manager (CM) to make new provisioning tags available.

load_localized_strings

> **Note:**
>
> Currently, localization is not supported in BRM.

Use this utility to load localized strings into the BRM database. This utility reads localized strings from various customizable files and stores them as **/strings** objects in the BRM database. These files include error code files, locale description files, reason code files, and a variety of other files.

> **Note:**
>
> Use only one locale for each type of file.

For information on modifying and loading localized string files, see "Localizing and Customizing Strings".

> **Caution:**
>
> When loading reason codes from the **reasons.***locale* file, **load_localized_strings** also loads information from this file into the **/config/map_glid** object. If customized to specify services and event types for event-level adjustments, the utility also loads information into the **/config/reason_code_scope** object. While the utility doesn't overwrite existing strings in the **/strings** object unless you direct it to, it *does* overwrite the **/config/reason_code_scope** and **/config/map_glid** objects.

> **Note:**
>
> To connect to the BRM database, the **load_localized_strings** utility needs a configuration file in the directory from which you run the utility. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

*BRM_home***/data/config**

Syntax

```
load_localized_strings [-v] [-f] [-h] filename.locale
```

Parameters

-v
Displays information about successful or failed processing as the utility runs.
load_localized_strings *any_other_parameter* **–v >** *filename***.log**

-f
Forces strings to be stored in the BRM database, overwriting localized strings with the same IDs. If you do not use **-f**, string objects are not stored when localized strings with the same IDs already exist.

> **Note:**
>
> This parameter has no effect on either the **/config/reason_code_scope** or **/config/map_glid** object. These objects are *always* completely overwritten by the utility.

-h
Displays help about using the **load_localized_strings** utility.

filename.locale
The name and location of the file that contains the localized strings.

- For sample BRM files **errors**, **locale_desc**, and **reasons**, see "Sample Files".
- locale is the BRM locale, based on ISO-639 and ISO-3166 standards.

> **Tip:**
>
> If you copy the *filename.locale* file to the same directory from which you run the **load_localized_strings** utility, you do not have to specify the path or the file name.

Results

If the **load_localized_strings** utility doesn't notify you that it was successful, look in the **load_localized_strings.log** file to find any errors. The log file is either in the directory from which the utility was started, or in a directory specified in the configuration file.

Sample Files

Use the following American English files as examples of how to set up localized or customized error message, locale description, and reason code files. The following files are loaded into the BRM database when you install BRM:

- **errors.en_US** in *BRM_home***/sys/msgs/errorcodes**
- **locale_descr.en_US** in *BRM_home***/sys/msgs/localedescr**
- **reasons.en_US** in *BRM_home***/sys/msgs/reasoncodes**

load_pin_config_business_type

Self-Care Manager uses business type definitions to display the appropriate entry fields for an account.

Using the load_pin_config_business_type Utility

Use this utility to load updates to the business types defined in the *BRM_home*/data/config/pin_config_business_type file into the/config/business_type object in the BRM database. When you load new business type definitions, you overwrite the old /config/business_type object.

You must restart the Connection Manager (CM) after running this utility.

> **Note:**
>
> If /config/business_type isn't loaded, the value for PIN_FLD_BUSINESS_TYPE included in the/account object at account creation must either be zero or not included in the input flist. Otherwise, PCM_OP_CUST_POL_VALID_BILLINFO returns a validation error.

Examples of Business Type Definitions

You can append your own business type definitions to the end of the list of definitions provided in **pin_config_business_type**, or create a new definition file. The file must include a **0 "Unknown business type"** entry. Keep the current entries of **1 "Consumer"** and **2 "Business"** in place and append your new entry after them.

The format for each entry is an integer value plus an associated quoted string with a semicolon at the end of the statement. The length of the string is limited to 1024 characters. Multi-line string entries are valid as long as there is a closing quote before the carriage return and an opening quote on the following line, for example:

```
0    "Unknown business type";
1    "Consumer";
2    "Business";
3    "This is a valid quoted string entry "
     "that spans more than one line";
```

There can be only one quoted string associated with each integer. The string description is not used for validation, but provides a way to record the meaning of each integer value.

To add a business type definition, use a text editor to open the **pin_config_business_type** file and follow the guidelines provided in the comment section to add your new value to the end of the existing definitions. For example, for an employee account, you might want the Account Creation wizard to display a field for the CSR to enter an employee ID number. To add the new value to the/config/business_type object for an employee business type, add it to the end of the existing definitions in the **pin_config_business_type** file:

```
0    "Unknown business type";
1    "Consumer";
2    "Business";
3    "Employee";
```

Location

BRM_home/data/config

Syntax

```
load_pin_config_business_type [-d] [-v] [-?] filename
```

Parameters

-d
Writes error information for debugging purposes to the utility log file. By default, the file is located in the same directory as the utility and is called **default.pinlog**. You can specify a different name and location in the **Infranet.properties** file.

-v
Displays information about successful or failed processing as the utility runs.

-?
Displays the syntax and parameters for this utility.

filename
The file containing the business type definitions, typically, *BRM_home***/data/config/pin_config_business_type**.

Results

Table 58-2 lists the possible completion values this utility returns. The returned value is saved in the **default.pinlog** file:

Table 58-2 load_pin_config_business_type Returned Values

Value	Description
0	success
1	bad flag
2	error parsing input file (default is **pin_config_business_type**)
3	error opening PCM connection
4	error opening transaction
5	error deleting old **/config/business_type** object
6	error creating new **/config/business_type** object
7	error committing transaction

load_pin_device_permit_map

Use this utility to load device-to-service mapping information into the BRM database.

You use a mapping file to define which service types can be associated with a particular device type. A sample file is provided as *BRM_home***/sys/data/config/pin_device_permit_map**.

After defining device-to-service mappings, you use the **load_pin_device_permit_map** utility to load the mapping data into a **/config/device_permit_map** object.

> **Note:**
>
> You must load the mapping data into the database and restart the CM before you can use device management features. Because device management configuration data is always customized, it is not loaded during BRM installation.

The **load_pin_device_permit_map** utility shares a configuration file with the **load_pin_device_state** utility. This file is generated during installation and is located in *BRM_home***/apps/device_management/Infranet.properties**.

See "Connecting BRM Utilities" in *BRM System Administrator's Guide* for more information about configuration files.

Location

*BRM_home***/bin**

Syntax

```
load_pin_device_permit_map [-v] [-d] map_file
```

Parameters

-v
Displays information about successful or failed processing as the utility runs.

-d
Writes error information for debugging purposes to the utility log file. By default, the file is located in the same directory as the utility and is called **default.pinlog**. You can specify a different name and location in the **Infranet.properties** file.

map_file
The name and location of the file that contains the device-to-service mapping data. A sample file is supplied as *BRM_home***/sys/data/config/pin_device_permit_map**. You can modify this file or create a new one.
The file includes lines specifying the device type (such as **/device/number**) and the service types (such as **/service/telco/gsm/telephony**) that can be associated with it. In this example, four service types can be associated with **/device/number**.

```
/device/number    :/service/telco/gsm/telephony
                  :/service/telco/gsm/sms
                  :/service/telco/gsm/fax
                  :/service/telco/gsm/data
```

The sample file includes additional information about the correct syntax for entries.

Results

If the **load_pin_device_permit_map** utility doesn't notify you that it was successful, look in the log file (normally **default.pinlog**) for error messages. The log file is located in the directory from which the utility was started, or in a directory specified in the **Infranet.properties** configuration file.

load_pin_device_state

Use this utility to load state-change definitions for devices into the BRM database.

You use a state change definition file to define, which state changes are valid and which policy opcodes to call for each state change. A sample file is provided as *BRM_home***/sys/data/config/pin_device_state**.

After defining the state changes, you run the **load_pin_device_state** utility to load the definitions into the database as a **/config/device_state** object.

> **Note:**
>
> You must load the state change definitions into the database and restart the Connection Manager before you can use device management features. Because device management configuration data is always customized, it is not loaded during BRM installation.

You create a single **/config/device_state** object for that device type by using the **root** login.

The **load_pin_device_state** utility shares a configuration file with the **load_pin_device_permit_map** utility. This file is generated during installation and is located in *BRM_home***/apps/device_management/Infranet.properties**.

When you enter state change information into the state change definition file, you must refer to the localized text strings that describe the valid states for a particular device type. You must define these text strings and load them into the database by using the **load_localized_strings** utility. A sample localized strings file for device states is located in *BRM_home***/sys/msgs/device_states/device_states.en-US**.

Location

*BRM_home***/bin**

Syntax

```
load_pin_device_state [-v] [-d] state_change_file
```

Parameters

-v
Displays information about successful or failed processing as the utility runs.

-d
Writes error information for debugging purposes to the utility log file. By default, the file is located in the same directory as the utility and is called **default.pinlog**. You can specify a different name and location in the **Infranet.properties** file.

state_change_file
The name and location of the state change definitions file, where *device* represents the device type. For example, the default file for Number Management is **pin_device_state_num**. Each device type must have its own file.
A sample state change definition file is provided in *BRM_home***/sys/data/config/pin_device_state**.
Use this syntax for entries:

```
object_type

device_type

state_id: state_type: strid_id: string_ver:opcode_num:flags
next_id1: opcode_num1:flags1
next_id2: opcode_num2:flags2
```

- **object_type**

  A subclass of **/config/device_state**, for example **/config/device_state/sim**. The object type must be the first non-comment line in the file. Only one object type can be specified.

- **device_type**

  The device type, for example **/device/sim**. The device type must be the second non-comment line in the file. Only one device type can be specified.

- **state_id**

  An integer state ID, such as **0**, **1**, **2**, and so on. State IDs can be freely assigned, except for 0, which is reserved for the Raw state.

- **state_type**

  An integer representing the state type to which **state_id** belongs. The state type determines the valid behaviors of states of that type. There are four possible values:

  - **0** (RAW): Includes the state that marks the beginning of the device life cycle. Only one state can be of this type. States of type RAW can transition only to states of type INIT. Device objects cannot be saved in the Raw state.
  - **1** (INIT): Includes all the states in which the device can be saved immediately after its creation. States of type INIT can transition to states of type INIT, NORMAL, or END.
  - **2** (NORMAL): Includes all the working states between INIT and END. States of type NORMAL can transition to INIT states, other NORMAL states, and END states.
  - **3** (END): Includes all the terminal states of the life cycle. Devices cannot be transitioned from states of this type.

- **str_id**

  ID of the state's localized text string in **/string**. You must load these text strings into the database by using a text file and the **load_localized_strings** utility.

- **str_version**

  Version number of the state's localized text string in **/string**.

- **opcode_num**

  The opcode number of the first policy opcode to be called during a transition from **state_id**. Device opcode numbers are specified in the **device.h** file. This file is located in *BRM_home***/include/ops** directory.

- **flags**

  Flags to be used when calling the opcode specified by **opcode_num**. Flags are listed in the **pin_device.h** file and explained in the opcode documentation.

- **next_id***x*

  An integer state ID that specifies a device state to which **state_id** can transition. Any integer value is allowed except 0, which is reserved for the Raw state.

- **opcode_num***x*

  Specifies the policy opcode called just after the state change to **next_id***x* is complete. Device opcode numbers are specified in the **device.h** file. This file is located in *BRM_home***/include/ops** directory.

- **flags***x*

  Flags to be used when calling the opcode specified by the matching **opcode_num***x*. Flags are listed in the **pin_device.h** file and explained in the opcode documentation.

The sample state change definitions file includes additional information about the correct syntax for entries. Its location is *BRM_home***/sys/data/config/pin_device_state**.

Results

If the **load_pin_device_state** utility doesn't notify you that it was successful, look in the log file (normally **default.pinlog**) for error messages. The log file is located in the directory from which the utility was started, or in a directory specified in the **Infranet.properties** configuration file.

The utility fails if it detects duplicate state IDs in the state change definitions file. It also fails if the file contains more than one object or device type entry.

load_pin_excluded_logins

Use this utility to load a list of service types that identify users through alias names rather than service logins.

You specify which service types identify users through alias names in the *BRM_home***/sys/data/config/pin_excluded_logins.xml** file. You then load the file into the **/config/login_exclusion** object in the BRM database.

When you run the utility, the **pin_excluded_logins.xml** and **business_configuration.xsd** files must be in the same directory. By default, both files are in *BRM_home***/sys/data/config**.

After running this utility, you must stop and restart the Connection Manager (CM).

> **Note:**
>
> To connect to the BRM database, the **load_pin_excluded_logins** utility needs a configuration file in the directory from which it is run. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

*BRM_home***/bin**

Syntax

```
load_pin_excluded_logins [-v] [-d] [-t] [-h] xml_file
```

Parameters

-v
Displays information about successful or failed processing as the utility runs.

-d
Creates a log file for debugging purposes. Use this parameter for debugging when the utility seemed to run without error but the data was not loaded into the database.

-t
Runs the utility in test mode to validate the XML file. This parameter does *not* create, modify, or delete any entries in the **/config/login_exclusion** object.

> **Tip:**
>
> To avoid load errors based on XML content problems, run the utility with this option *before* loading data into the database.

-h
Displays help information for using this utility.

xml_file
The name and location of the XML file. The default XML file is *BRM_home***/sys/data/config/pin_excluded_logins.xml**, but the utility can take any XML file name as a parameter as long as the file's contents conform to the appropriate schema definition.
If you copy the file to the same directory from which you run the load utility, specify only the file name. If you run the command in a different directory from where the file is located, you must include the entire path for the file.

> **Note:**
>
> The file must be in the same directory as the **business_configuration.xsd** and **pin_excluded_logins.xml** files.

Validity Checks

The utility validates the XML file against rules defined in the **pin_excluded_logins.xsd** file. This file resides in the *BRM_home***/sys/data/config** directory, and you must point to it through a **business_configuration.xsd** file in the directory that contains your working XML file.

The utility validates the following:

- The service name length is from 1 to 1,023 characters.
- The service names listed in the file are unique.

Results

If the **load_pin_excluded_logins** utility doesn't notify you that it was successful, look in the log file (normally **default.pinlog**) for error messages. The log file is located in the directory from which the utility was started or in a directory specified in the **pin.conf** configuration file.

load_pin_order_state

Use this utility to load state-change definitions for orders into the BRM database.

You define which state changes are valid and which policy opcodes to call for each state change in a state change definition file. A sample file is provided as *BRM_home***/sys/data/config/pin_order_state**.

After defining the state changes, you use the **load_pin_order_state** utility to load the definitions into the database as a **/config/order_state** object.

> **Note:**
>
> You must load the state change definitions into the database and restart the CM before you can use order management features. Because order management configuration data is always customized, it is not loaded during BRM installation.

You create a single **/config/order_state** object for that order type by using the **root** login.

When you enter state change information into the state change definition file, you must refer to the localized text strings that describe the valid states for a particular order type. You must define these text strings and load them into the database by using the **load_localized_strings** utility. A sample localized strings file for order states is located in *BRM_home***/sys/msgs/order_states/order_states.en-US**.

Location

*BRM_home***/bin**

Syntax

```
load_pin_order_state [-v] [-d] state_file_order
```

Parameters

-v
Displays information about successful or failed processing as the utility runs.

-d
Writes error information for debugging purposes to the utility log file. By default, the file is located in the same directory as the utility and is called **default.pinlog**. You can specify a different name and location in the **Infranet.properties** file.

state_file_order
The name and location of the state change definitions file, where *order* represents the order type. For example, for Number Management, the file could be **pin_order_state_num**. Each order type must have its own file.
A sample state change definition file is provided in *BRM_home***/sys/data/config/pin_order_state**.
Use this syntax for entries:

```
object_type

order_type

state_id: state_type: strid_id: string_ver
 : next_id1
 : next_id2
```

- **object_type**

  A subclass of **/config/order_state**, for example **/config/order_state/sim**. The object type must be the first non-comment line in the file. Only one object type can be specified.

- **order_type**

  The order type, for example **/order/sim**. The order type must be the second non-comment line in the file. Only one order type can be specified.

- **state_id**

  An integer state ID, such as 0, 1, 2, and so on. State IDs can be freely assigned, except for 0, which is reserved for the Raw state.

- **state_type**

  An integer representing the state type to which **state_id** belongs. The state type determines the valid behaviors of states of that type. There are four possible values:

  - **0** (RAW): Includes the state that marks the beginning of the order life cycle. Only one state can be of this type. States of type RAW can transition only to states of type INIT. Order objects cannot be saved in raw state.
  - **1** (INIT): Includes all the states in which the order can be saved immediately after its creation. States of type INIT can transition to states of type INIT, NORMAL, or END.
  - **2** (NORMAL): Includes all the working states between INIT and END. States of type NORMAL can transition to INIT states, other NORMAL states, and END states.
  - **3** (END): Includes all the terminal states of the life cycle. Orders cannot be transitioned from states of this type.

- **str_id**

  ID of the state's localized text string in **/string**. You must load these text strings into the database via a text file and the **load_localized_strings** utility.

- **str_version**

  Version number of the state's localized text string in **/string**.

- **next_id***x*

  An integer state ID that specifies a order state to which **state_id** can transition. Any integer value is allowed except 0, which is reserved for Raw state.

The sample state change definitions file includes additional information about the correct syntax for entries. Its location is *BRM_home***/sys/data/config/pin_order_state**.

Results

If the **load_pin_order_state** utility doesn't notify you that it was successful, look in the log file (normally **default.pinlog**) for error messages. The log file is located in the directory from which the utility was started, or in a directory specified in the **Infranet.properties** configuration file.

The utility fails if it detects duplicate state IDs in the state change definitions file. It also fails if the file contains more than one object or order type entry.

parse_custom_ops_fields

Use this BRM Perl script to parse include files for custom fields and opcodes and to generate memory-mappable files that extend the opcode and field name-to-number mapping tables. This allows you to use custom fields and opcodes by using their name or number in applications.

> **Note:**
>
> You can define opcodes and fields without including them in the mapping tables; however, you cannot use them in client applications or **testnap** by using their symbolic names instead of numbers.

The **parse_custom_ops_fields** script reads the specifications of opcodes and fields from the *input* header file and creates corresponding entries in the *output* memory-mapped file or header file.

The script also generates a Java class for each field. You must compile each class with **JavaPCM.jar** in the **CLASSPATH**. You can either include the **CLASS** files in a **JAR** file, or build them in a base directory and leave them there. Then you must add the **JAR** file or the base directory to the **CLASSPATH**.

If you build the class files in the base directory, make sure the base directory matches the package name. For example, if the *java_package* is **com.portal.classFiles**, then the base directory must be **/com/portal/classFiles**.

For custom fields, the script creates a properties file in the *java_package* directory, named **InfranetPropertiesAdditions.properties**. You must append this file to your **Infranet.properties** file.

Location

*BRM_home***/bin**

Syntax

```
parse_custom_ops_fields -L language -I input -O output -P java_package
```

Parameters

language
The BRM API used. It can be **pcmc** (for PCM C), or **pcmjava** (for Java PCM).

> **Note:**
>
> **perlpcmif** is a wrapper API for PCM C. If you run the script with the **pcmc** option, you can call custom opcodes in your Perl PCM-based client applications.

input
The header file you create for your custom opcodes and fields.

output
The memory-mapped file or directory for the output of the script. If *language* is **pcmjava**, then *output* must be a directory having some correspondence with the Java package. For example, if the *java_package* is in **com.portal.classFiles**, then *output* must be **f:/mysource/com/portal/classFiles**.

java_package
The Java package, where you want to put the generated classes.

pin_adu_validate

Use this utility to dump and validate information for one or more accounts from the BRM database.

For more information on using **pin_adu_validate** utility, see "Validating Account Data" in *BRM Managing Customers*.

> **Note:**
>
> To connect to the BRM database, this utility needs a configuration file in the directory from which you run it. The **pin.conf** file for this utility is in *BRM_home***/sys/diagnostics/pin_adu_validate**. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

*BRM_home***/sys/diagnostics/pin_adu_validate**

Syntax

```
pin_adu_validate [-dump [-validate]] [-report]
```

Parameters

-dump
Uses the account search flist in the input file configured in the **pin_adu_validate** configuration file (**pin.conf**) to search for objects associated with the accounts in the BRM database and dumps the object data into an output file.

-validate
Performs the predefined validations enabled in the **pin_adu_validate** configuration file (**pin.conf**) file and any custom validations defined in the PCM_OP_ADU_POL_VALIDATE policy opcode.

> **Note:**
>
> To perform validation, you must specify both the **-dump** and **-validate** options at the same time.

-report
Searches for account information in the BRM database using the account search flist in the input file configured in the **pin_adu_validate** configuration file (**pin.conf**) file and provides statistical data about the accounts, such as the number of object instances found for each object specified in the **pin.conf** file. The statistical data is written to the Connection Manager (CM) log file.

> **Note:**
>
> **pin_adu_validate** uses the date ranges configured in the **pin.conf** file to provide statistics for most commonly updated objects. See "Limiting Dump Information by Specifying a Date Range" in *BRM Managing Customers*.

Sample output:

```
Number of /account object instances found for the account [82828]: 1
Number of /service/email object instances found for the account [82828]: 1
Number of /service/ip object instances found for the account [82828]: 1
Number of /payinfo/cc object instances found for the account [82828]: 1
```

pin_bus_params

Use the **pin_bus_params** utility to retrieve and load configurable business parameters for the **/config/business_params** objects in the BRM database. These parameters enable optional BRM features or control things like the tracking level for write-off reversals, whether to validate exclusions for discounts, and so forth.

You use this utility to perform two tasks:

- Retrieve the contents of a **/config/business_params** object and convert its contents into XML for easy modification.
- Load a modified XML file containing a parameter class and its associated parameters into the appropriate **/config/business_params** object in the BRM database.

The utility retrieves the **/config/business_params** objects, converts them to XML, and writes them into the *BRM_home***/sys/data/config/pin_bus_params**_*ParameterClassName***.xml.out** file. The utility also loads the objects into BRM from this file, converting them back into the format required by the object. You can optionally place the file in a different location, but if you do, you must also copy the **bus_params_conf.xsd** file from the *BRM_home***/xsd** directory to the new location and modify the file to include the correct relative paths to the **bus_params**_*ParameterClassName***. xsd** files.

You can use another utility, **pin_cfg_bpdump**, to dump the contents of all business parameters in XML format. You can direct the XML output to a file or to a utility or application. See "Dumping Business Parameters in XML Format" in *BRM System Administrator's Guide*.

> **Caution:**
>
> When loading business parameters, the **pin_bus_params** utility overwrites the **/config/business_params** object for the parameter class that appears in the XML file. If you are updating some of the parameters in the class, you cannot load the new parameters only. You must load a complete set of parameters for the class.

The **pin_bus_params** utility is a perl script. To run the utility, you *must* set path to perl in your environment.

> **Note:**
>
> To connect to the BRM database, the **pin_bus_params** utility needs a configuration file in the directory from which you run the utility. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

*BRM_home***/bin**

Syntax

For retrieving a **/config/business_params** object for a parameter class:

```
pin_bus_params [-h] -r ParameterClassTag bus_params_ParameterClassName.xml
```

For loading a **/config/business_params** object from a specified file:

```
pin_bus_params [-h] bus_params_ParameterClassName.xml
```

Parameters

-r
Retrieves the contents of a **/config/business_params** object and converts it to XML for editing.

-h
Displays the syntax and parameters for this utility.

ParameterClassTag
Specifies the parameter class tag for the parameter class you are retrieving. This parameter is case sensitive and uses the following naming convention:
BusParams*ObjectClassName*
Where *ObjectClassName* is the name of the class in the **/config/business_params** object. For example, the object class name for the **/config/business_params** object that contains the business parameters that control billing is **billing**, so the parameter class tag is **BusParamsBilling**. This parameter is case sensitive.

bus_params_ParameterClassName.xml
If you use the **- r** parameter, this parameter specifies the name and location of the XML output file created by the utility. This file contains the business parameters for the class identified in the *ParameterClassName* part of the file name.
If you do not use the **- r** parameter, this parameter specifies the name and location of the XML input file that you are loading into the **/config/business_params** object. This file contains the business parameters for the class identified in the *ParameterClassName* part of the file name. BRM overwrites the **/config/business_params** object that class.
To specify a different directory, include the full path and file name, as in the following example that loads XML file contents into the **/config/business_params** object for the **billing** business parameters:

```
pin_bus_params C:\param_conf\bus_params_billing.xml
```

> **Note:**
>
> If you are loading parameters from a directory other than *BRM_home***/sys/data/config**, the directory you are using must contain a **business_configuration.xsd** file. The utility uses the **bus_params**_*ParameterClassName***.xsd** file to validate the XML file.

Validity Checks

The utility checks XML validity against rules defined in the **bus_params**_*ParameterClassName***.xsd** file. This file resides in the *BRM_home***/sys/data/config** directory, and you must point to it through a **bus_params_config.xsd** file in the directory that contains your working XML file.

The validity check ensures that the XML meets these standards:

- The parameter class name in the file is unique.
- The parameter class contains at least one parameter.

Results

The **pin_bus_params** utility notifies you when it successfully creates or modifies the **/config/business_params** object. Otherwise, look in the **default.pinlog** file for errors. This file is either in the directory from which the utility was started, or in a directory specified in the utility configuration file.

If you use the utility to load a **/config/business_params** object, you can display the object by using the Object Browser, or use the **robj** command with the **testnap** utility. See "Reading an Object and Writing Its Contents to a File". This example shows an element in the **/config/business_params** object:

```
0 PIN_FLD_POID             POID [0] 0.0.0.1 /config/business_params 10830 0
0 PIN_FLD_CREATED_T      TSTAMP [0] (1083892760) Thu May 06 18:19:20 2004
0 PIN_FLD_MOD_T          TSTAMP [0] (1083892760) Thu May 06 18:19:20 2004
0 PIN_FLD_READ_ACCESS       STR [0] "G"
0 PIN_FLD_WRITE_ACCESS      STR [0] "S"
0 PIN_FLD_ACCOUNT_OBJ      POID [0] 0.0.0.1 /account 1 0
0 PIN_FLD_DESCR             STR [0] "Business logic parameters for AR"
0 PIN_FLD_HOSTNAME          STR [0] "-"
0 PIN_FLD_NAME              STR [0] "ar"
0 PIN_FLD_PROGRAM_NAME      STR [0] "-"
0 PIN_FLD_VALUE             STR [0] ""
0 PIN_FLD_VERSION           STR [0] ""

...

0 PIN_FLD_PARAMS          ARRAY [2] allocated 4, used 4
1    PIN_FLD_DESCR          STR [0] "Enable/Disable payment suspense management.
                                     The parameter values can be 0 (disabled),
                                     1 (enabled). Default is 0 (disabled)."
1    PIN_FLD_PARAM_NAME     STR [0] "payment_suspense_enable"
1    PIN_FLD_PARAM_TYPE     INT [0] 1
1    PIN_FLD_PARAM_VALUE    STR [0] "1"
```

> **Note:**
>
> To connect to the BRM database, you must restart the Connection Manager (CM) to activate new business parameters.

pin_cfg_bpdump

Use the **pin_cfg_bpdump** utility to dump BRM business parameters in XML format. You can direct the output to a file or to another utility or application, such as a diagnostic application.

For more information, see "Dumping Business Parameters in XML Format" in *BRM System Administrator's Guide*.

> **Note:**
>
> To connect to the BRM database, the **pin_cfg_bpdump** utility needs a configuration file in the directory from which you run the utility. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

BRM_home/**diagnostics/pin_cfg_bpdump**

Syntax

```
pin_cfg_bpdump
```

Parameters

This utility has no parameters.

Results

The **pin_cfg_bpdump** utility outputs in XML format the contents of all **/config/business_params** objects.

The **pin_cfg_bpdump** utility does not produce **pinlog** notifications of success or failure. If there is an error, the utility produces no output. You can refer to the CM and DM logs for information about the problem.

pin_crypt_app

Use this utility to encrypt files and plaintext passwords, generate encryption keys, load keys into memory, store root keys in Oracle wallets, and automatically modify the Oracle Data Manager (DM) **pin.conf** file. This utility supports both the OZT and AES encryption methods.

For more information, see:

- Encrypting Data
- About AES Encryption
- About Encrypting Passwords

Location

BRM_home/**bin**

Syntax

To encrypt files and plaintext passwords in AES:

```
pin_crypt_app  -enc [-f Filename]
```

To generate an encrypted AES key:

```
pin_crypt_app  -genkey [-key Key]
```

To generate a 256-bit encrypted key from an AES key:

```
pin_crypt_app  -genkey [-key AES_key]
```

To store configuration data in the wallet:

```
pin_crypt_app -setconf -wallet clientWalletLocation -program programName -parameter
configEntry -value value
```

To retrieve configuration data from the client wallet:

```
pin_crypt_app -getconf -wallet clientWalletLocation -program programName -parameter
configEntry
```

To generate or modify a root key:

```
pin_crypt_app  -genrootkey
```

To encrypt files and plaintext using the Oracle ZT PKI algorithm:

```
pin_crypt_app  -useZT -enc [-f Filename]
```

Parameters

-enc [-f *Filename*]
Encrypts files, plaintext, and plaintext passwords in AES format. When **-enc** is used alone, the utility prompts you for the password or plaintext to encrypt.
Include the **-f** *Filename* option to encrypt the specified file.

-genkey
Generates a 256-bit encrypted AES key. BRM generates a random AES key internally to generate the encrypted AES key. Use one or both of these options with the parameter:

- **-key**: The utility generates a 256-bit encrypted AES key. Key is the 256-bit key in hexadecimal notation (64 hexadecimal characters).
- **-update_dm_conf**: The utility adds the new key to the *BRM_home***/sys/dm_oracle/pin.conf** file.

If this is the first time that an AES key has been created, stop and restart the DM when prompted by the utility.

-key *AES_key*
Generates a 256-bit encrypted key from the specified AES key. Use this parameter if you already have an AES key and do not want BRM to generate one internally.

-genrootkey
Generates or modifies a root key and stores it in a root key wallet. The root key is used for encryption using the Oracle ZT PKI–approved encryption algorithm.
For more information on the Oracle ZT PKI encryption algorithm, see "Configuring the Data Manager for Oracle ZT PKI Encryption".

-useZT
Uses the Oracle ZT PKI algorithm to encrypt files and plaintext and to generate a key. Use the following options with this parameter:

- **-enc -f** *Filename*: The utility uses the Oracle ZT PKI encryption algorithm to encrypt the specified file. *Filename* is the name of the file to encrypt.
- **-enc***:* After prompting the user for the plaintext, the utility uses the Oracle ZT PKI encryption algorithm to encrypt the specified text.
- **-genkey -key** *Key*: The utility uses the Oracle ZT PKI encryption algorithm to generate a 256-bit encrypted key. *Key* is the 256-bit key in hexadecimal notation (64 hexadecimal characters).

-wallet *clientWalletLocation*
Specifies the path to the Oracle wallet for setting or getting the configuration values.

-parameter *ConfigEntry*
Specifies the configuration entry in the wallet that you want to set or get the values for.

-value *value*
Specifies the value that must be set in the Oracle wallet.

-program
Specifies the name of the program that is storing the configuration entry, such as **dm**.

-help
Displays the syntax and parameters for this utility.

Results

The **pin_crypt_app** utility returns the output when the operation is successful. It does not return errors.

pin_deploy

Transport definitions for storable classes and fields from one BRM server to another. This utility reads storable class and field definitions from a Portal Object Definition Language (PODL) file and loads them into the BRM data dictionary. This utility also exports storable class and field definitions from the BRM data dictionary to a PODL file.

> **Note:**
>
> Before attempting to create new storable classes, verify that enough space is available in the BRM database. This utility can not test for available space. If the available space is exhausted before deployment is complete, new storable classes might be in an inconsistent state.

See "Deploying Custom Fields and Storable Class Definitions".

Location

BRM_home/**bin**

Commands

There are five commands for **pin_deploy**:

- Verify
- Create
- Replace
- Class
- Field

To print the syntax and parameters for this utility, type **-h** or **-help**.

Verify

Connect to BRM server, determine changes to be made, and report any conflicts. May alternatively accept PODL from **stdin**.

```
pin_deploy verify [file_one file_two ... file_N]
```

Example:

```
pin_deploy verify myobj.podl
```

Connects to the default database schema specified in the **pin.conf** file, determines the changes required for creating the storable class and field definitions contained in the PODL file, and reports conflicts.

Verify for dm_invoice

If your BRM installation includes a separate **dm_invoice** database, you can use two parameters with the **verify** command:

- Use the **-d** switch to connect to a **dm_invoice** database for a BRM installation that initially used a **dm_oracle** database schema to store **/invoice** objects. If you use the **-d** switch, specify the target database schema by database number. Omit the **-d** switch and database number to connect to the default database schema.
- Use the **-e** switch to print debugging information to the log file.

  pin_deploy verify [**-de**] [*target_db*] [*file_one file_two ... file_N*]

Example:

```
pin_deploy verify -de 0.0.6.1 myobj.podl
```

Connects to the **dm_invoice** database 0.0.6.1, determines the changes required for creating the storable class and field definitions contained in the PODL file, reports conflicts, and prints debugging information to the log file.

Create

Load storable class and field definitions into the data dictionary. Succeeds only if there are no conflicts. If there are conflicts, they are reported and no action occurs. May alternatively accept PODL from **stdin**.

pin_deploy create [*file_one file_two ... file_N*]

Example:

```
pin_deploy create myobj.podl
```

Connects to the default database schema specified in the **pin.conf** file, creates the storable class and field definitions contained in the PODL file, and prints debugging information to the log file. If conflicts are encountered, the operation fails without taking any action.

Create for dm_invoice

If your BRM installation includes a separate **dm_invoice** database, you can use two parameters with the **create** command:

- Use the **-d** switch to connect to a **dm_invoice** database for a BRM installation that initially used a **dm_oracle** database schema to store **/invoice** objects. If you use the **-d** switch, specify the target database schema by database number. Omit the **-d** switch and database number to connect to the default database schema.
- Use the **-e** switch to print debugging information to the log file.

  pin_deploy create [**-de**] [*target_db*] [*file_one file_two ... file_N*]

Example:

```
pin_deploy create -de 0.0.6.1 myobj.podl
```

Connects to the **dm_invoice** database 0.0.6.1, creates the storable class and field definitions contained in the file, and prints debugging information to the log file. If conflicts are encountered, the operation fails without taking any action.

Replace

Load storable class and field definitions into the data dictionary. Overwrites storable class and field definitions even if conflicts exist. The SQL table and column names for storable classes can not be overwritten. When loading field definitions, only the field description attribute is overwritten. May alternatively accept PODL from **stdin**.

```
pin_deploy replace [file_one file_two ... file_N]
```

Example:

```
pin_deploy replace myobj.podl
```

Connects to the default database schema specified in the **pin.conf** file, creates the storable class and field definitions contained in the PODL file, and prints debugging information to the log file. If conflicts occur, existing definitions are overwritten.

Replace for dm_invoice

If your BRM installation includes a separate **dm_invoice** database, you can use two parameters with the **replace** command:

- Use the **-d** switch to connect to a **dm_invoice** database for a BRM installation that initially used a **dm_oracle** database schema to store **/invoice** objects. If you use the **-d** switch, specify the target database schema by database number. Omit the **-d** switch and database number to connect to the default database schema.
- Use the **-e** switch to print debugging information to the log file.

  ```
  pin_deploy replace [-de] [target_db] [file_one file_two...file_N]
  ```

Example:

```
pin_deploy replace -de 0.0.6.1 myobj.podl
```

Connects to the **dm_invoice** database 0.0.6.1, creates the storable class and field definitions contained in the PODL file, and prints debugging information to the log file. If conflicts occur, existing definitions are overwritten.

Class

Export storable class definitions from a BRM server in PODL format. Can specify any number of storable classes on command line. If no storable classes are specified, then all storable classes are exported.

```
pin_deploy class [-mnscp]  [class_one  class_two ... class_N]
```

[-m] Export storable class implementation.

[-n] Export storable class interface.

[-s] Include all subclasses of specified storable class.

[-c] Include field definitions for all customer-defined fields within storable classes.

[-p] Include field definitions for all BRM-defined fields within storable classes.

Examples:

```
pin_deploy class -mn /account /bill
```

Export definitions for the **/account** and **/bill** storable classes from a BRM server in PODL format. Includes both implementations and interfaces.

```
pin_deploy class -s /event
```

Export the **/event** storable class interface along with all of its subclasses.

Field

Export field definitions from a BRM server in PODL format. May specify any number of fields by name. If no fields are specified, then all fields will be exported unless the **-c** or **-p** parameters are used.

```
pin_deploy field [-cp] [field_one field_two ... field_N]
```

[-c] Include field definitions for all customer-defined fields.

[-p] Include field definitions for all BRM-defined fields.

Examples:

```
pin_deploy field PIN_FLD_PRODUCTS PIN_FLD_NAMEINFO
```

Export definitions for the PIN_FLD_PRODUCTS and PIN_FLD_NAMEINFO fields from a BRM server in PODL format.

```
pin_deploy field -cp
```

Export definitions for all customer and BRM-defined fields from a BRM server in PODL format.

pin_uei_deploy

Use this utility to migrate event import templates from one BRM database schema to another. You use event import templates to load data from log files into BRM as billable events. For more information, see "Migrating Event Import Templates from One BRM Database to Another" in *BRM Loading Events*.

> **Note:**
>
> To connect to the BRM database, the **pin_uei_deploy** utility needs a configuration file in the directory from which you run the utility. See "Connecting BRM Utilities" in *BRM System Administrator's Guide*.

Location

BRM_home/**bin**

Syntax

```
pin_uei_deploy   -l|-c|-m|-d|-r|-v
                 -t template name -i input_file -o output_file [-h]
```

Parameters

-t *template_name*
The name of the template. This parameter is followed by the name of the event import template that you *read*, *create*, *delete*, or *modify* (depending on the command).

-i *input_file*
The input file (event import template) to load into the database schema.

-o *output_file*
The output file to save on the local system.

-h
Displays the syntax and parameters for this utility.

Commands

- List
- Create
- Modify
- Delete
- Read
- Verbose

List

Lists all event import templates stored in the database schema.

```
pin_uei_deploy -l
```

There are no parameters for this command.

Create

Creates the specified event import template.

```
pin_uei_deploy -t template_name -c -i output_file
```

Example:

Create an event import template named **CDR3** and load it into the database schema:

```
pin_uei_deploy -t CDR3 -c -i CDRoutput
```

> **Note:**
>
> If an event import template with the same name exists in the database schema, the operation fails and **pin_uei_deploy** logs an error. Delete the existing event import template first, or overwrite it by using the modify operation.

Modify

Modifies the specified event import template.

```
pin_uei_deploy -t template_name -m -i output_file
```

Example:

Load into the database schema an event import template called **CDR3** and overwrite the existing template with the same name:

```
pin_uei_deploy -t CDR3 -m -i CDRoutput
```

Delete

Deletes the specified event import template.

```
pin_uei_deploy -t template_name -d
```

Example:

Delete the event import template named **CDR3**:

```
pin_uei_deploy -t CDR3 -d
```

Read

Reads the specified event import template from the database schema.

```
pin_uei_deploy -t template_name -r -o output_file
```

Example:

Read the event import template named **CDR3**, and save it to an output file on the local system named **CDRoutput**:

```
pin_uei_deploy -t CDR3 -r -o CDRoutput
```

Verbose

Displays information about successful or failed processing as the utility runs.

```
pin_uei_deploy -v any_other_parameter
```

pin_virtual_time

Use this utility to adjust or display BRM's current time and date, without affecting the operating system time and date. This utility is useful for testing billing and other time-sensitive functions in BRM.

For information about using the **pin_virtual_time** utility, see "Testing Your Price List" in *BRM Setting Up Pipeline Pricing*.

> **Caution:**
>
> Use **pin_virtual_time** only with a test database. You should not change the BRM system time on a production BRM database.

> **Note:**
>
> To test custom client applications that are connected to the CM, you can use PCM_OP_GET_PIN_VIRTUAL_TIME to get the virtual time that is set by **pin_virtual_time**.

Operation

To run BRM with **pin_virtual_time** enabled:

1. Make sure all BRM components are stopped.
2. Configure all BRM components, via their **pin.conf** files, to use **pin_virtual_time**. A file containing time and date information for the **pin_virtual_time** utility is created the first time that **pin_virtual_time** is run. BRM recommends you designate *BRM_home***/lib/***pin_virtual_time_file* with the **-f** parameter.
3. Type **pin_virtual_time** with the **-m** option to set the mode and value of the time.

   > **Note:**
   >
   > If there are multiple BRM machines, run **pin_virtual_time** on all of them.

4. Start all BRM components.
5. Perform testing as desired.
6. Between testing stages, adjust the time with **pin_virtual_time**. You can change modes.
7. After completing testing, stop all BRM components
8. Remove or comment out (with #) the **pin_virtual_time** entry in the **pin.conf** files.
9. Perform database cleanup if needed.

Dependencies

All BRM server component **pin.conf** files must contain the following line to use **pin_virtual_time** to set BRM's time:

```
- - pin_virtual_time BRM_home/lib/pin_virtual_time_file
```

The *pin_virtual_time_file* file contains the information that BRM requires to determine the time/date mode and how to set it.

The *pin_virtual_time_file* file is mapped into memory by **pin_virtual_time** when each BRM component starts up. If different BRM components are running on different machines (for example, the Connection Manager on one machine and the Data Manager on another), then **pin_virtual_time** must be enabled in the **pin.conf** files on both machines, and whenever **pin_virtual_time** is set on one machine it must be set correspondingly on the other machine(s). Failure to do so may cause BRM to operate incorrectly.

See "Using Configuration Files" in *BRM System Administrator's Guide* for information on setting up **pin.conf** files.

> **Note:**
>
> Make sure installation is complete before you put a **pin_virtual_time** line in the **pin.conf** file for your DM.

Syntax

```
pin_virtual_time [-i interval] [-m mode [time_value] [-y]]
                 [-f filename] [-h | -H | -?]
```

Parameters

By default (without the **-m** option), **pin_virtual_time** prints the current **pin_virtual_time** to *stdout* once and then exits.

-i *interval*
Print the current BRM time every *interval* seconds to *stdout* (until interrupted by CTRL C).

-m *mode* [*time_value*] [-y]
Set BRM according to *mode* and *time_value*:

mode
0 = Use operating system time (normal mode). BRM uses operating system time with no adjustments.
1 = Use *time_value* as a constant time (freeze mode). Time is frozen at the specified time until the **pin_virtual_time** command is used again to change the time or mode. *Use only when absolutely necessary, because BRM expects time to be moving.*
2 = Use *time_value* as the new time, and keep the clock running (offset mode). Time is adjusted to the time specified, and then advances one second every second. This is the mode that should be used for testing.

time_value
Use the format *MMDDHHMM*[*CC*]*YY*[.*SS*].

-y
Accept backwards movement. Allow the specified time to be before the current **pin_virtual_time** (time can be moved backwards).

> **Caution:**
>
> Move time backwards only when rebuilding BRM from scratch. Otherwise, moving time backwards can cause severe data corruption.

-f *filename*
Store the **pin_virtual_time** structure in the designated file and location. BRM recommends *BRM_home***/lib/***filename*. This path and file name must match the path and file name specified in the **pin_virtual_time** line in the **pin.conf** files for each BRM component.

-h, -H, -?
Displays the syntax and parameters for this utility.

Results

If the utility doesn't notify you that it was successful, look in the utility log file (**default.pinlog**) to find any errors. The log file is either in the directory from which the utility was started, or in a directory specified in the configuration file.

Examples

Print the current **pin_virtual_time** setting and mode:

```
% pin_virtual_time
mode 2  940102477  Sat Oct 16 12:34:37 1999
```

Print current **pin_virtual_time** every four seconds:

```
% pin_virtual_time -i 4
mode 2  940102527  Sat Oct 16 12:35:27 1999
mode 2  940102531  Sat Oct 16 12:35:31 1999
mode 2  940102535  Sat Oct 16 12:35:35 1999
mode 2  940102539  Sat Oct 16 12:35:39 1999
^C
%
```

Set **pin_virtual_time** to offset mode 12/31/98 11:30:43:

```
% pin_virtual_time -m 2   123111301998.43
filename BRM_home/lib/pin_virtual_time_file, mode 2, time: Thu Dec 31 11:30:43 1998
```

Set **pin_virtual_time** to normal mode:

```
% pin_virtual_time -m 0
filename BRM_home/lib/pin_virtual_time_file, mode 0
```

testnap

The **testnap** utility enables a developer to manually interact with the BRM server by establishing a PCM connection with the Connection Manager (CM) and then executing PCM operations by using that connection. See "Using the testnap Utility to Test BRM".

Dependencies

The **testnap** utility requires a **pin.conf** configuration file to connect to your BRM system. You can either use the CM's **pin.conf** file, or run **testnap** from a directory that contains a suitable **pin.conf** file, such as *BRM_home***/sys/test**.

The **testnap** utility relies on POID logins and types to identify the specific account or storable class to modify. A POID database number is required as a placeholder only.

The correct path to the shared libraries must also be configured. The library path is configured by default to point to the standard libraries in *BRM_home***/lib**. If the libraries are not in *BRM_home***/lib**, then you must set the library path environment appropriately.

Syntax

```
testnap
command [args]
```

About Buffer Numbers

The **testnap** utility allocates numerous internal buffers, which are used to store object or flist fields. Buffers are referenced by integers of the user's choice. Every time a new buffer is referenced, **testnap** allocates that new buffer.

- If you do not specify a buffer number for a command that expects one, you will be prompted for a buffer number.
- The **meta** keyword causes **testnap** to display the size of external buffer fields. By default, the contents of external buffer fields are displayed.

Testnap Commands

- **r** {<<token [file]} [buf]

  Read an flist or object from a file and put it in a **testnap** buffer. The <<*token* operator causes **testnap** to read from **stdin** until the token string is read.

- **r+** {<<*token* [*file*]} [*buf*]

  Read and append flist to existing **buf**. The <<*token* operator causes **testnap** to read from **stdin** until the token string is read.

- **w** [*buf*] [*file*]

  Write contents of **testnap** buffer to a file.

- **w+** [*buf*] [*file*]

  Append contents of a **testnap** buffer to the same file.

- **l**

  List the **testnap** buffers that are currently in use.

- **d** [*buf*]

  Display a **testnap** buffer.

- **!** *cmds* [*args*]

  Run a shell command (for Linux systems only).

- **s** [*buf*]

  Save an flist or object from the input buffer to a **testnap** buffer.

- **<** [*file*]

  Run cmd script.

- **p** [<*property*> <*value*>]

  Display or set properties.

- **q**

  Quit this program.

- **h**, **help**,**?**

  Print usage help message.

- **create** {[*buf*] | [**poid**]}

  Create an object. Run PCM_OP_CREATE_OBJ and print the return flist to **stdout**. The poid keyword causes the poid id of the object that is created to be the poid id specified in the input flist.

- **delete** {[*buf*] | [- <*db*> <*type*> <*id*>]}

  Delete an object. Run PCM_OP_DELETE_OBJ and print the return flist to **stdout**. The poid of the object to be deleted can be specified on the command line.

- **robj** {[*buf*] | [- <*db*> <*type*> <*id*>]} [**meta**]

  Read an object. Run PCM_OP_READ_OBJ using either the poid in the flist in buf, or the poid specified on the command line. Prints the return flist (the contents of the object) to **stdout**.

- **rflds** *buf* [**meta**]

  Read fields. Run PCM_OP_READ_FLDS using the flist in **buf**, and print the return flist to **stdout**. Each field (or row) in the field list must be in a valid flist format. The values for the last field are arbitrary, but must be valid for their type. For example, you have to include **""** for **STR** fields and (some number) for a **TSTAMP** field. If either of these fields are blank, an error is returned.

- **wflds** [*buf*]

  Write fields. Run PCM_OP_WRITE_FLDS and print the return flist to **stdout**.

- **dflds** [*buf*]

  Delete fields. Run PCM_OP_DELETE_FLDS and print the return flist to **stdout**.

- **search** *buf* [**meta**] [**count**]

  Search. Run PCM_OP_SEARCH and print the return flist to **stdout**. The **count** operator sets the PCM_OPFLG_COUNT_ONLY flag, which causes **search** to return only the number of matches found by the search. The count is returned as the **ELEM_ID** of the **RESULTS** array on the output flist.

- **ssrch** *buf* [**meta**]

  Step-search. Run PCM_OP_STEP_SEARCH and print the return flist to **stdout**.

- **snext** *buf* [**meta**]

  Step-search next. Run PCM_OP_STEP_NEXT to get the next object in a step search, and print the return flist to **stdout**.

- **send** *buf*

  End step-search. Run PCM_OP_STEP_END and print the return flist to **stdout**.

- **gdd** {[*buf*] | [- <*db*> <*type*> <*id*>]}

  Get data dictionary. Run PCM_OP_GET_DD and print the return flist to **stdout**. The poid can be specified on the command line.

- **sdd** < *flags*> [*buf*]

  Set data dictionary. Run PCM_OP_SET_DD and print the return flist to **stdout**.

- **sort** *buf sort_buf* [*descending_flag*]

  Read contents of **buf**, sort it non-recursively using the template in **sort_buf**, and print the sorted flist to **stdout**. Available on Linux only. The descending flag is optional, with **0** (the default) indicating ascending order, and any non-zero integer indicating descending order.

  Sorting is not implemented for the following:

  – POID
  – BINSTRS
  – BUFFERs

  - ERRBUFS

    The number of ARRAY elements in the sort specification is ignored. Use **0**. Sort specifications just need valid numbers. The numbers are not necessarily valid or desired values for the current sort; they are basically required place-holders.

- **rsort** *buf sort_buf* [*descending_flag*]

  Recursive sort. Read contents of **buf**, sort it recursively using the template in **sort_buf**, and print the sorted flist to **stdout**. Available on Linux only. The descending flag is optional, with **0** (the default) indicating ascending order, and any non-zero integer indicating descending order.

- **open** [**ro**|**rw**|**lock** {[*buf*] | [- <*db*> <*type*> <*id*>]}]

  Open transaction. Run PCM_OP_TRANS_OPEN and print the return flist to **stdout**.

- **commit**

  Commit the current transaction. Run PCM_OP_TRANS_COMMIT and print the return flist to **stdout**.

- **abort**

  Cancel the current transaction. Run PCM_OP_TRANS_ABORT and print the return flist to **stdout**.

- **inc** [*buf*]

  Increment one or more fields of an object. Run PCM_OP_INC_FLDS and print the return flist to **stdout**.

- **noop** [*buf*]

  Run non-operational opcode. Run PCM_OP_TEST_LOOPBACK on that database schema and print the return flist to **stdout**.

- **pass** [*buf*]

  Run a pass_thru op to server. Run PCM_OP_PASS_THRU, an extension op that just sends an flist to a DM that supports it.

- **xop** *op flag buf*

  Run opcode *op* with flags set to *flag*, the contents of **buf** as the input flist, and print the sorted flist to **stdout**.

- **id**

  Print user and session ID.

- **echo** *string*

  Echo a string.

Error Handling

If an error occurs, the contents of the error buffer (**ebuf**) which corresponds to the error are written to **stderr**.

Error Example 1

```
# Delete attempt with the - argument missing:
pin@demo5-668> testnap
===> database 0.0.0.1 from pin.conf "userid"

delete 0.0.0.1 /account 1
```

```
ERROR: bad number "0.0.0.1"
no object to use for delete
```

Error Example 2

```
# Attempt to read a non-existent object:

robj - 0.0.0.1 /account 11988

PCM_OP_READ_OBJ failed: err 3:PIN_ERR_NOT_FOUND, field 0/16:PIN_FLD_POID,
       loc 4:PIN_ERRLOC_DM, errclass 4:PIN_ERRCLASS_APPLICATION, rec_id 0, resvd 30001
```

For more information on the error buffer, see "Finding Errors in Your Code".

Examples

For an extensive set of examples, refer to the *Examples* section of "Using the testnap Utility to Test BRM".

pin_config_editor

Use the **pin_config_editor** utility to set or get configuration entries for JAVA PCM client applications from the client wallet.

Location

BRM_home/**bin**

Syntax

To store a value in the wallet:

pin_config_editor -setconf -wallet *clientWalletLocation* **-parameter** *configEntry* **-value** *value*

To store a password in the wallet:

pin_config_editor -setconf -wallet *clientWalletLocation* **-parameter** *configEntry* **[-pwd]**

To retrieve configuration data from the client wallet:

pin_config_editor -getconf -wallet *clientWalletLocation* **-parameter** *configEntry*

To view configuration entries in the client wallet:

pin_config_editor -list -wallet *clientWalletLocation*

To display the syntax and parameters for this utility:

pin_config_editor [**-help**]

Parameters

-setconf
Adds or modifies values.

-getconf
Gets values.

-wallet *clientWalletLocation*
Specifies the path to the Oracle wallet for setting or getting the configuration values.

-parameter *ConfigEntry*
Specifies the configuration entry in the wallet that you want to set or get the values for.

-value *value*
Specifies the value that must be set in the Oracle wallet.

-pwd
Prompts you for the password to be stored in the client wallet. The password that you enter does not appear on the screen. If you want to store the encrypted password, enter the encrypted password at the command prompt.

-list
Displays all configuration entries in the wallet.

-help
Displays the syntax and parameters for this utility.

Results

The **pin_config_editor** utility sets the specified configuration entry and lets users retrieve configuration values from the client wallet.

generate_config_xml_from_opcode_spec.py

Use the **generate_config_xml_from_opcode_spec.py** script to generate a loadable version of an opcode specification for an updated BRM opcode or a custom opcode. For example, you can modify the maximum length of a BRM opcode flist field or create a loadable opcode specification for a custom opcode.

For more information, see "Modifying the Maximum Length of an Opcode Flist Field".

Location

BRM_home/**sys/data/config**

Syntax

generate_config_xml_from_opcode_spec.py *opcodeSpecXmlFile*

Parameters

opcodeSpecXmlFile

The XML file name of the opcode specification for an existing BRM opcode or a custom opcode, for example: **PCM_OP_CUST_SET_LOGIN.xml**.

Result

The script generates a file named **config**_*opcodename*.**xml**, where *opcodename* is the opcode name minus PCM_OP_. For example, if *opcodeSpecXmlFile* is named **PCM_OP_CUST_COMMIT_CUSTOMER.xml**, the script generates an output file named **config_CUSTOM_COMMIT_CUSTOMER.xml**.

> **Note:**
>
> To enable the opcode to use its new specification, you must load the XML output file into the database using **load_config**.