Oracle® Communications Network Charging and Control

SDK Developer's Guide

Release 12.0.2

E95924-01

December 2018

ORACLE

Oracle Communications Network Charging and Control SDK Developer's Guide, Release 12.0.2

E95924-01

Contents

8 Creating Memory-Mapped Files

9 Creating and Replicating Database Tables

10 Creating an EDR Loader Plugin

11 Creating a CcsAuth Voucher PAM Plugin

Preface

This guide describes how to use the software development kit (SDK) for the Oracle Communications Network Charging and Control (NCC) software. The SDK is based on the NCC application programming interface (API) and allows you to create NCC components that satisfy customer-specific requirements that the NCC software does not address.

Audience

This document is intended for software developers who are consultants or system integrators tasked with addressing customer requirements within the NCC framework.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

About Customizing Network Charging and Control

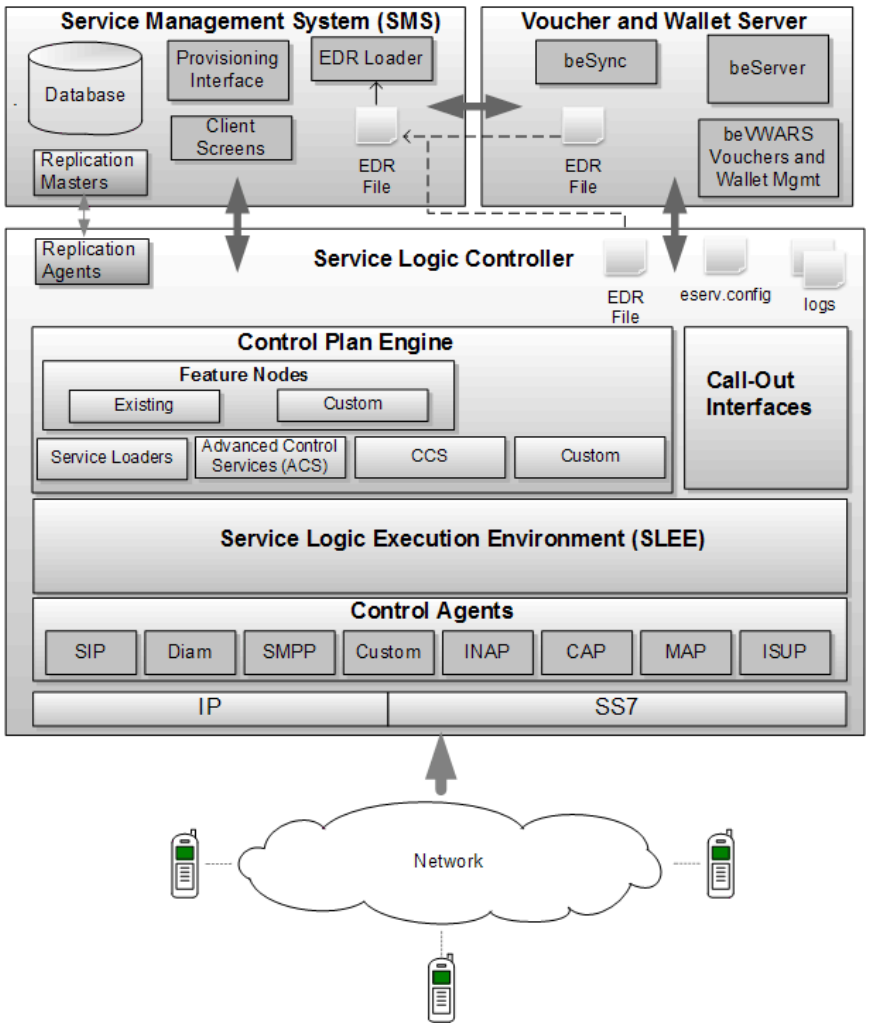
This chapter gives an overview of the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

Understanding the SDK Development Environment

The Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK) provides C++ Application Programming Interfaces (APIs) and supporting files and tools that enable you to develop custom NCC components and features and integrate them into the core NCC product.

Figure 1–1 illustrates the major elements of the Network Charging and Control system.

Figure 1–1 NCC System Elements



The three major NCC components are the Service Management System (SMS), the Service Logic Controller (SLC), and the Voucher and Wallet Server (VWS).

The SMS performs these tasks:

- System management, including centralized management of alarms, logs, and reporting
- Service management, including centralized configuration of products, control plans, and rates
- Customer relationship management (CRM) and provisioning, including centralized management of subscribers and wallets
- Voucher management, including provisioning and management of vouchers and voucher batches
- Replication management, including replication of service, subscriber, wallet and voucher information to other nodes in the NCC platform, including the SLC and the Voucher and Wallet Server (VWS).

The SLC includes the following components:

- The Service Logic Execution Environment (SLEE), which manages a group of applications that can communicate with each other and share resources efficiently
- The Advanced Control Services (ACS), which is the real-time engine for control plan execution
- Charging drivers, which are SLEE interfaces that provide connectivity to the VWS, Billing and Revenue Management (BRM), or third-party online charging systems
- The Control Agent layer, which is a framework for SLEE interfaces that connect the SLC to the core network

  Control agents communicate with ACS using the Intelligent Network Application Part (INAP) language. In doing so, they translate network protocols such as SIP, MAP, or Diameter into INAP so that a common service logic is possible, independent of the network protocol.

  A control agent issues an InitialDP request to initiate a dialog for a particular event or session.
- Service Logic Layer, which includes feature nodes, chassis actions and service loaders for charging, messaging and number services

The VWS handles prepaid rating, balance management, voucher management, and promotion management. It includes the following key components:

- The beSync component, which synchronizes reservations, wallets, and vouchers between VWS nodes.
- The beServer component, which handles incoming requests and distributes them to beVWARS processes.
- The beVWARS component, which performs all business logic in the VWS, including rating, recharging, promotions and voucher redemption. The beVWARS process is the engine of the VWS.

Developing NCC Components and Features with the SDK

The SDK enables you to develop the following types of components and features to satisfy your specific NCC requirements:

- Service loaders

  Developing a custom service loader allows you to manipulate information contained in the InitialDP according to your requirements and allows you to load a control plan according to your own business rules. The InitialDP is the INAP request that a control agent, or external network element, sends to slee_acs to trigger processing for a new event or session – for example, for a call, SMS, or data session. The slee_acs is the Advanced Control Service process that runs on the Service Logic Controller and processes service logic, generally by executing a control plan.

  The purpose of a service loader is to prepare slee_acs for the execution of service logic for the service associated with a new event or session. For example, for a chargeable mobile voice call, the service loader typically would:

  – Lookup the calling party number in the subscriber database
  – Retrieve the subscriber profile for use in the control plan
  – Retrieve all service configuration data for that subscriber, including the product, tariff plans, and so on.

  - Determine the details of wallets associated with the subscriber for charging and tracking
  - Set slee_acs up for execution of the control plan for charging a mobile originating voice call

  Furthermore, you can use a service loader to closely control the information returned by the CONNECT operation.

  You can also develop a custom service loader that extends an existing product-based service by supplying one or more functions that are called after the ones in the standard product library. This is known as a service loader extender.

  For more information, see "Creating Service Loaders".

- Feature nodes

  You can create a custom feature node, which is invoked by a control plan, to perform customer-specific processing that is not included by the NCC system feature nodes. A custom feature node can retrieve information from a specific database table, read from and write to profiles, set up charging information in a vendor-specific form, and perform various Intelligent Network Application Part (INAP) operations.

  For more information, see "Creating a Custom Feature Node".

- Control agents

  A control agent provides a bridge between a particular protocol and the service logic being run by slee_acs. A custom control agent allows you to interface the SLC with network elements that use protocols that are not supported by the SLC.

  For more information, see "Creating a Custom Control Agent".

- Provisioning interface (PI) commands

  Allow you to provision information that is stored in custom tables. Such tables could contain routing data that is specific to the topology of your network. You can develop Man-Machine Language (MML), Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP)-based PI commands.

  For more information, see "Creating Provisioning Interface Commands".

- Provisioning screens

  Allow you to create screens to enter and maintain data in new database tables that you create.

  For more information, see "Creating Provisioning Screens".

- Memory-mapped files (Mfiles)

  Allow you to cache in memory, information from database tables that could contain routing data specific to your topology. This is useful, for example, when directly accessing these tables could have a negative impact on performance. Mfiles provide rapid data retrieval through the runtime service flow with negligible impact on performance.

  For more information, see "Creating Memory-Mapped Files".

- Event Detail Record plugins

  An event data record (EDR) loader plugin processes an EDR produced by a VWS. The VWS produces an EDR for every change to a wallet or voucher. An EDR loader process on the SMS processes each EDR.

The core purpose of the EDR loader is to load EDRs into the database so customer care agents can read them in the screens. You can develop custom plugins, for example, to format an EDR specifically for a third party billing system, or convert an EDR into Oracle's Billing and Revenue Management (BRM) SQL42 format.

For more information, see "Creating an EDR Loader Plugin".

- CcsAuth Voucher PAM plugins

  A CcsAuth Voucher PAM plugin provides an implementation of a Voucher Pluggable Authentication Module. This handles generation of the voucher secret (HRN) which is written to the voucher print file and how the hashed or encrypted HRN (private secret) is then created and stored in the database.

  For more information, see "Creating a CcsAuth Voucher PAM Plugin"

About the SDK API

The NCC SDK is built on the NCC API, which provides a library of classes and files that enable you to create NCC components and features to meet your specific NCC system requirements.

The NCC API consists of global classes and classes that are grouped namespaces listed in Table 1–1, which provide scope for them and their member functions and definitions:

Table 1–1 NCC API Namespaces

Namespace	Description
acs	Contains all APIs related to the ACS software. See ncc and ncc:acs namespaces also.
acsActionsAPI	Contains classes that interact with the Advanced Control Service to request discreet actions associated with making a call and processing results. Actions include getting the wallet, requesting an initial reservation, getting rates, setting the discount, getting subscriber credit card details, and so on.
ccs	Contains all APIs for the CCS software component.
ccs:cdr	Contains classes and the interface for CDR loader plugins.
ccs::auth	Contains classes and the interface for CCS Voucher PAM plugins.
cmn	Contains the common, or generally-applicable, functions and classes related to supporting features and functions.
cmn::cfg	Defines configuration information object and values.
ncc	Contains acs, inap, slee, and tcap namespaces.
ncc::acs:	Stores ACR charging flags; throws unknown language exception; creates and handles ACS notifications; defines a single slee_acs transaction; gets a generic request and returns a generic response
ncc::inap	Creates INAP operation, indication, and request primitives.
ncc::slee	In general, provides functionality for passing messages across the SLEE, between applications and interfaces. In particular, provides a layer of abstraction, so that custom code can pass information between processes without having to worry about handling the SLEE transport layer.
ncc::tcap	Contains classes that define the TCAP standard primitives.

Table 1–1 (Cont.) NCC API Namespaces

Namespace	Description
sms	Contains the APIs for the SMS software component.
sms::pi	Contains the APIs for the Provisioning Interface.
sms::pi::common	Contains classes that define error codes related to logging on and session startup, request processing, XML, and internal errors.

For information on accessing the NCC API documentation, see "Accessing the API Documentation".

2

Getting Started

This chapter explains how to install and get started with the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK) and describes the prerequisites.

Prerequisites

The NCC SDK is delivered in a zip file that also includes this guide. The general platform requirements for the SDK are the same as they are for NCC. For more information on the general platform requirements for NCC, see the *Installation Guide.*

Before using the SDK, you should have working knowledge and skills in the following areas:

- The C, C++, and Java programming languages
- The Unix operating system, including system programming and system administration
- Structured Query Language (SQL) for managing databases in a relational database management system, particularly Oracle
- The following NCC concepts:
  - Advanced Control Services (ACS) system architecture
  - The role of ACS in a service flow
  - The configuration of the Service Management System (SMS) product features

In addition, the SDK requires the following software and tools for building components:

- For Oracle Linux:
  - Linux 7 update 1

    The NCC build environment must support the minimum version of the environment on which the software will be deployed and run. For more information, see the following web page regarding Oracle's binary guarantee:

    http://www.oracle.com/technetwork/server-storage/solaris/overview/guarantee-jsp-135402.html
  - Oracle Database 12*c* Release 1 ((Oracle 12.1.0)), including client libraries

    The NCC build environment must support the minimum version of the environment on which the software will be deployed and run.

Requires installing 32-bit client libraries. The client libraries are installed in ORACLE_HOME/lib32

  - Oracle Java 1.8.0_161
  - GNU Compiler Collection (gcc) 4.8.3, included with C++
  - GNU Binary Utilities (binutils) 2.23.2; binutils are ported to most major Unix versions
  - GNU make (gmake) 3.82, included with C++
  - GNU Bison (bison) 2.7, included with C++
- For Oracle Solaris:
  - Solaris 11.3

    The NCC build environment must support the minimum version of the environment on which the software will be deployed and run. For more information, see the following web page regarding Oracle's binary guarantee:

    http://www.oracle.com/technetwork/server-storage/solaris/overview/guarantee-jsp-135402.html
  - Oracle Database 12*c* Release 1 ((Oracle 12.1.0)), including client libraries

    The NCC build environment must support the minimum version of the environment on which the software will be deployed and run.

    Requires installing 32 bit client libraries. The client libraries are installed in ORACLE_HOME/lib32
  - Oracle Java 1.8.0_161
  - GNU Compiler Collection (gcc) 4.8.2, built from source with GNU linker ld. See "Building gcc 4.8.2 on Oracle Solaris With GNU Linker ld."
  - GNU Binary Utilities (binutils) 2.23.2; built from source. See "Building binutils 2.23.2 on Oracle Solaris."
  - GNU make (gmake) 3.82, included with C++
  - GNU Bison (bison) 2.3, included with C++

See the product documentation for these tools for information on installing and using them.

Building gcc 4.8.2 on Oracle Solaris With GNU Linker ld

To build gcc 4.8.2 on Solaris with GNU Linker ld:

1. Run the following commands to set the environment variables:

   ```
   export PATH=/opt/ncctools/bin:${PATH}
   export LD_LIBRARY_PATH=/opt/ncctools/lib:${LD_LIBRARY_PATH}
   ```

2. Run the following commands:

   ```
   sudo mkdir /scratch/username
   sudo chown username:dba /scratch/username
   sudo mkdir /opt/ncctools
   ```

   where *username* is the user name with the relevant permissions.

3. Create a temporary directory (**gcc**).

4. Download **gcc-4.8.2.tar.gz** for Solaris from https://ftp.gnu.org/gnu/gcc/gcc-4.8.2/ to **gcc**.
5. Run the following command to unzip the **gcc-4.8.2.tar.gz** file.

   ```
   tar zxvf ../gcc-4.8.2.tar.gz
   ```

6. Go to **gcc** and run the following commands to build gcc 4.8.2:

   ```
   export AR=/usr/gnu/bin/ar
   export AS=/usr/gnu/bin/as
   export LD=/usr/gnu/bin/ld
   export NM=/usr/gnu/bin/nm
   mkdir gcc-4.8.2-obj
   cd gcc-4.8.2-obj
   ../gcc-4.8.2/configure --prefix=/opt/ncctools --with-gnu-ar
   --with-ar=/opt/ncctools/bin/ar --with-gnu-as --with-as=/opt/ncctools/bin/as
   --with-gnu-ld --with-ld=/opt/ncctools/bin/ld --with-gnu-nm
   --with-nm=/opt/ncctools/bin/nm --enable-languages=c,c++,fortran,objc
   --enable-shared --with-gmp-include=/usr/include/gmp
   --with-mpfr-include=/usr/include/mpfr CFLAGS='-g -O2  -mtune=ultrasparc
   -mcpu=ultrasparc -mno-unaligned-doubles' CXXFLAGS='-g -O2 -mtune=ultrasparc
   -mcpu=ultrasparc -mno-unaligned-doubles'
   gmake bootstrap
   sudo gmake install
   unset AR
   unset AS
   unset LD
   unset NM
   ```

Building binutils 2.23.2 on Oracle Solaris

To build binutils 2.23.2 on Solaris:

1. Run the following commands to set the environment variables:

   ```
   export PATH=/opt/ncctools/bin:${PATH}
   export LD_LIBRARY_PATH=/opt/ncctools/lib:${LD_LIBRARY_PATH}
   ```

2. Run the following commands:

   ```
   sudo mkdir /scratch/username
   mkdir binutils
   cd binutils
   ```

   where *username* is the user name with the relevant permissions.

3. Download **binutils-2.23.2.tar.gz** for Solaris from https://ftp.gnu.org/gnu/binutils/ to **binutils**.
4. Run the following command to unzip the **binutils-2.23.2.tar.gz** file.

   ```
   tar zxvf ../binutils-2.23.2.tar.gz
   ```

5. Run the following commands to build binutils 2.23.2:

   ```
   mkdir binutils-2.23.2-obj
   cd binutils-2.23.2-obj
   ../binutils-2.23.2/configure --prefix=/opt/ncctools --disable-werror
   gmake
   sudo gmake install
   ```

Installing the SDK

To install the SDK, you must extract the package to a directory and use the Unix **tar** command to extract the contents of the **.tar** file into the specified directory. For example, the following commands unzip the package and extract the contents of the **.tar** file into the current directory.

```
gunzip SDK-12.0.2.0.0.tar.gz
tar -xf SDK-12.0.2.0.0.tar
```

Setting Environment Variables

You must specify the location of the SDK files by adding commands to your **.profile** file to set the NCC_SDK_HOME environment variable and then export it. For example, the following lines specify the location of the SDK as being **/home/***username***/sdk**, where *username* varies and represents the name of a user's home directory.

```
NCC_SDK_HOME=/home/username/sdk
export NCC_SDK_HOME
```

In addition, you should set the environment variables described in Table 2–1:

Table 2–1 Environment Variable Settings for SDK

Environment Variables	Value
LD_LIBRARY_PATH	Specify the **/lib** directory for **binutils** and **gcc** before anything else. For example: ■ On Solaris, **/opt/ncctools** ■ On Linux, **/usr/local/lib**
PATH	Specify the **/bin** directory for **binutils** and **gcc** before anything else. For example: ■ On Solaris, **/opt/ncctools** ■ On Linux, **/usr/local/lib**
ORACLE_HOME	Specify the absolute path to Oracle 12*c* home. For example: **/u01/app/oracle/product/12.1.0**

SDK Contents

The SDK installation process creates the following set of directories:

- **bin**: Contains utility programs such as the database table installer, the feature node installer, and the PI command installer.
- **doc**: Contains the HTML files that document the application. For more information on the contents of this folder, see "Accessing the API Documentation".
- **example**: Contains example files for each of the components that you can build with the SDK. For more information on the contents of this folder, see "Building Examples".
- **include**: Contains the SDK header files that you will need to include in your applications.

  For information about which files to include to have access to a particular function or class, see the examples or the API reference. For information on accessing the API reference, see "Accessing the API Documentation".

- **jar**: Contains SDK Java archive files that contain the classes to develop custom screens.
- **lib**: Contains a collection of static and dynamically linked libraries for use in the applications that you develop. The following list indicates which libraries need to be linked for each type of custom component:
  - **libcmnUtils.so** - any process that requires access to the Oracle database
  - **libcmnUtilsNoOra.so** - any process that does not require access to the Oracle database
  - **libcmnConfig.a** - any component that reads configuration files
  - **libcmnConfigFileImpl.a** - any component that reads configuration files
  - **libsmsStats.a** - any component that records statistics
  - **libSleeApiCallContext.so** - service loaders or macro nodes that use service-specific data
  - **libPI_common.so** - PI commands
  - **ibsleeDispatcher.so** - control agents and call-out interfaces
  - **libSLEE.so** - control agents and call-out interfaces
  - **libacsSleeTransaction.so** - control agents
  - **libtcapSleeTransaction.so** - TCAP control agents
  - **libgenericEvent.so** - call-out interfaces
  - **libgenericSleeTransaction.so** - call-out interfaces
  - **libcmnMfile.a** - an Mfile daemon or component that uses Mfile API
  - **libshare.a** - an Mfile daemon or component that uses Mfile API
  - **libccsAuthPluginSDK.so** - Ccs Auth Voucher PAM plugins
- **make**: Contains the **makefile** file, which defines the generic rules for creating each of the example components. Use it to compile the examples. The additional files, such as **install**, **install.sh**, and **env.mk,** are used by **makefile**. The **install.sh** file, for example, copies custom libraries, once they are built, to **$NCC_SDK_HOME/lib**.

Building Examples

Follow these steps to build the examples that are provided with the SDK:

1. Change location to directory **$NCC_SDK_HOME/example/***example_dir* where *example_dir* is the name of the directory that holds the particular examples that you want to build.
2. Enter the following command to build the examples:

```
gmake install
```

The `gmake install` command builds all of the examples in the specified directory.

The SDK includes the examples listed in Table 2–2:

Table 2–2 SDK Examples

Example Directory	Examples
ABC/java	Sample SMS screen, built into **abc.jar**
ABC/db	Definitions for sample database tables, which can be used with **cmnTableInstaller.sh**
sdkCDRLoaderPlugin	Sample EDR loader plugin, which is built into the **sdkCDRLoaderPlugin.so** shared library. Also contains **sdkCDRLoaderPlugin.h** and **Makefile** files
sdkCallout	Sample call-out SLEE interface, **sdkCallout**
sdkCommon	Common utilities used by feature nodes and service loaders
sdkMacroNodes	Sample macro nodes, which are built into the **sdkMacroNodes.so** shared library file and loaded by slee_acs. Also contains feature node database definitions in **sdkMacroNodes.xml**, which can be used with **acsMacroNodeInstaller**
sdkMfileAPI	Sample Mfile access library, which is build into **libsdkMfileAPI.a**
sdkMfileDaemon	Sample Mfile daemon, **sdkMfileDaemon**
sdkPiCommands/SDK_ABCNOA	Sample PI command, built into **libPI_SDK_ABCNOA.so**, loaded by **PImanager.** Definitions for sample PI commands, which can be used with **PICommandInstaller**.
sdkServiceLoader/generalExample	Sample generic service loader, built into **libsdkServiceLoader.so**, loaded by slee_acs
sdkServiceLoader/example1	Sample service loader, built into **libsdkServiceExample1.so**, loaded by slee_acs
sdkServiceLoader/example2	Sample service loader, built into **libsdkServiceExample2.so**, loaded by slee_acs
sdkServiceLoader/example3	Sample service loader, built into **libsdkServiceExample3.so**, loaded by slee_acs
sdkServiceLoader/example4	Sample service loader, built into **libsdkServiceExample4.so**, loaded by slee_acs
sdkTcapAgent	Sample TCAP control agent, **sdkTcapAgent**
sdkccsAuthPlugin	Sample CCS Auth plugin, which is built into the **libsdkCcsAuthPlugin.so** shared library. Also contains sdkCcsAuthPlugin.hh and Makefile files.

Installing the Examples

After building the examples, follow these steps to install them:

1. On the SLC, create the following directory:

   /IN/service_packages/SDK

2. Copy the example applications in Table 2–3 to the specified directory on the SLC:

Table 2–3 SLC Install Locations for Examples

Files in Build Location	Install Location
$NCC_SDK_HOME/lib/*.so	**/IN/service_packages/SDK/lib**
$NCC_SDK_HOME/bin/sdkCallout **$NCC_SDK_HOME/bin/sdkTcapAgent** **$NCC_SDK_HOME/bin/sdkMfileDaemon**	**/IN/service_packages/SDK/bin**

3. Copy the example applications listed in Table 2–4 to the specified directory on the SMS:

Table 2–4 SMS Install Locations for Examples

Files in Build Location	Install Location
$NCC_SDK_HOME/bin/abc.jar.sig	**/IN/html**
$NCC_SDK_HOME/bin/acsMacroNodeinstaller	**/IN/service_packages/ACS/bin**
$NCC_SDK_HOME/bin/ccsAuthPluginInstaller	**/IN/service_packages/CCS/bin**
$NCC_SDK_HOME/bin/pluginschema.dtd	**/IN/service_packages/SDK/etc**
$NCC_SDK_HOME/bin/nodeschema.dtd	**/IN/service_packages/SDK/etc**
$NCC_SDK_HOME/example/sdkMacroNodes/ **sdkMacroNodes.xml**	**/IN/service_packages/SDK/etc**
$NCC_SDK_HOME/bin/cmnTableInstaller.sh **$NCC_SDK_HOME/bin/cmnTableInstaller_temp_** **install.sh** **$NCC_SDK_HOME/bin/cmnTableInstaller_temp_** **uninstall.s**h **schema.dtd** **client.dtd**	**/IN/service_packages/SDK/bin**
$NCC_SDK_HOME/example/ABC/db/SDK.xml **$NCC_SDK_HOME/example/ABC/db/SDK_** **Client.xml**	**/IN/service_packages/SDK/bin**
$NCC_SDK_HOME/bin/PICommandInstaller **$NCC_SDK_HOME/bin/CommandsSchema.dtd**	**/IN/service_packages/SDK/bin**
$NCC_SDK_HOME/example/ **sdkPiCommands/SDK_ABCNOA/SDK_PI.xml**	**/IN/service_packages/SDK/bin**

4. With the SDK mounted on the SMS via NFS at $NCC_SDK_HOME, run the following commands:

```
cd /IN/service_packages
mkdir SDK
mkdir SDK/bin
mkdir SDK/etc
chgrp -R esg /IN/service_packages/SDK
chmod -R 750 /IN/service_packages/SDK

cp $NCC_SDK_HOME/bin/cmnTableInstaller*.sh SMS/bin
cp $NCC_SDK_HOME/bin/cmnDBInstallGenerator.jar SMS/bin
cp $NCC_SDK_HOME/bin/schema.dtd SDK/etc
cp $NCC_SDK_HOME/bin/client.dtd SDK/etc
cp $NCC_SDK_HOME/example/ABC/db/SDK.xml SDK/etc
cp $NCC_SDK_HOME/example/ABC/db/SDK_Client.xml SDK/etc

cp $NCC_SDK_HOME/bin/acsMacroNodeInstaller ACS/bin
```

```
cp $NCC_SDK_HOME/bin/ccsAuthPluginInstaller CCS/bin
cp $NCC_SDK_HOME/bin/nodeschema.dtd SDK/etc
cp $NCC_SDK_HOME/bin/pluginschema.dtd SDK/etc
cp $NCC_SDK_HOME/example/sdkMacroNodes/sdkMacroNodes.xml SDK/etc
```

5. Run the following commands to create the custom database tables and replication on the SMS:

```
$ su - smf_oper
$ cmnTableInstaller.sh -U smf -D ../SDK/install -S ../SDK/etc/SDK.xml -C
../SDK/etc/SDK_Client.xml
$ ../SDK/install/SMS/scripts/install.sh
```

> **Note:** The XML file either needs to be in the same directory as the DTD file, or you must modify it to specify the DTD file location.

6. Run the following commands to define the custom nodes on the SMS:

```
su - acs_oper
acsMacroNodeInstaller -install ../SDK/etc/sdkMacroNodes.xml
```

> **Note:** The XML file either needs to be in the same directory as the DTD file, or you must modify it to specify the DTD file location.

7. Run the following commands to define the custom Voucher PAM plugins on the SMS:

```
su - ccs_oper
ccsAuthPluginInstaller -i -f ../SDK/etc/sdkCcsAuthPlugin.xml
```

> **Note:** The XML file either needs to be in the same directory as the DTD file, or you must modify it to specify the DTD file location.

See "Creating a New Service Screen" for information on installing the example screens.

Accessing the API Documentation

The SDK API is described in a set of HTML files in the **doc** folder under **$NCC_SDK_HOME**.

To access the API reference documentation, which describes the NCC SDK classes and their members, open the following file using the browser of your choice:

$NCC_SDK_HOME/doc/html/index.html

Using Debugging, Alarms, Statistics, and Configuration

The NCC SDK includes supporting functions that enable you to add debug statements to your code, define and raise alarms, define and record statistics, and access the configuration file.

Using Debugging Statements

To turn on all of the debugging output for any program, run the following commands before starting the program:

```
DEBUG=all
export DEBUG
```

For slightly less output, you can run the following commands instead. The std flag sets all debug flags except for those having the highest volume.

```
DEBUG=std
export DEBUG
```

You can add debug statements throughout your code to assist in tracing and debugging.

To implement debugging in your code, do the following:

1. Include the **cmnDebugSDK.h** file.
2. Choose a debug section by defining the DEBUG_SECTION variable in the Makefile.
3. Call the cmnDebug_SECTION_FLAG() function to make this debug section available.
4. Add IDOUT and DOUT statements to write debug output. The IDOUT statement indents the subsequent output stream until the end of the current scope.

The following code illustrates these steps and writes out the name of the function and a line of debug output:

```
#include <cmnDebugSDK.h>

cmnDebug_SECTION_FLAG();

someFunction() {
    IDOUT << "someFunction()" << std::endl;

    // Do something
    ...

    // Log some debug output
    DOUT << "Did something" << std::endl;
}
```

SDK debug output is distinguished from NCC product output with a prefix of SDK, as shown in the following example:

```
acsEngine.c   503 [PID]       Engine   acsEngineProcessCall: Call macro node
processor
customFile.cc   4 [PID]   SDKsection    someFunction()
customFile.cc  20 [PID]   SDKsection    Did something
```

The two lines output from `SDKsection` require that the following line has been added to the Makefile:

```
DEBUG_SECTION=section
```

The following sections provide additional details.

Using Debug Sections

The SDK API implements the concept of debug sections, which marks all debug messages, either explicitly or implicitly, with a DEBUG_SECTION flag. The DEBUG_ SECTION flag allows you to turn on or turn off debug output at runtime by debug

sections. For example, you could turn on debug output only for the section `ACS_Chassis`.

The software writes each line of debug to a particular section, which you have either defined explicitly or else using the DEBUG_SECTION make variable by default.

At runtime, the operator can control which lines of debug are actually output by setting the DEBUG environment variable. Each line of code that generates debug output will only write it out if the relevant section is switched on by way of the DEBUG environment variable.

The DEBUG environment variable is always read before programs start, that is, before the function main() is called. Consequently, you can set $DEBUG to a comma separated list of debug section names to turn on debug output for those sections. The following line shows an example:

```
DEBUG=misc, myDebugSection
```

When assigning debug section names, it is a good practice to use the format PROGRAM_SUBSYSTEM for the flag. For example, for the program ACS and the subsystem Chassis, use the flag ACS_Chassis, as in `DEBUG_SECTION=ACS_Chassis`.

If you want multiple subsections, you can append to the name, for example, ACS_Chassis_Config. For programs, use the program name, such as **acsStatsMaster**.

The macros in Table 2–5 might be of interest.

Table 2–5 Debug Macros

Macro	Description
cmnDebug_SECTION_INIT();	You must initialize C debug flags by hand because C does not allow global variables to be initialized to the results of a function call. Place this in main() or another function.
cmnDebug_INIT(name)	Use this (or cmnDebug_SECTION_INIT()) on HP-UX to initialize debug flags by hand. Global variables in shared libraries are not initialized when the initializer is a function call. Also use to initialize a debug section inside a function that gets called at startup time or in a shared library. Running cmnDebug_INIT(name) on a flag that is already initialized has no effect.
cmnDebug_FLAG(name)	Initialize a debug section inside C++ source files in global scope.
cmnDebug_USE()	To use another debug flag that is not declared in the current file. If you use this inside a C++ namespace, the flag must also be defined inside the same namespace.

You can override DEBUG_SECTION in a C or C++ source file as shown in the following example:

```
#undef DEBUG_SECTION
#define DEBUG_SECTION SMS_Replication_filenames
```

You can discover the debug sections declared in an executable or library by searching for the string `cmnDebug_FLAG`. The following commands show how to do this.

```
strings ./myApp | grep cmnDebug_FLAG
```

Creating Debug Output

To create debug output using the C++ cout style interface, use the DOUT and IDOUT statements as shown in the following example.

```
DOUT << "read " << numBytes << " bytes from " << myName << std::endl;
IDOUT << "Entering new code chunk: " << name << std::endl;
```

IDOUT indents all output lines until the end of scope. All output goes to the same place.

The following output statements allow you to override the DEBUG_SECTION.

```
SDOUT(ACS_Chassis_details) << "read " << numBytes << " bytes from " << myName <<
std::endl;
SIDOUT(ACS_Chassis_details) << "Entering new code chunk: " << name << std::endl;
```

Like IDOUT, SIDOUT indents all output lines until the end of scope.

Use these statements to produce hex dumps of memory.

```
DBGMEM(pointer, length);
SDBGMEM(ACS_INAP_Messages)(pointer, length);
```

`SDBGMEM` allows you to override the DEBUG_SECTION flag.

Using Display Options

By default, the debugging output statements prefix each line with the current date and time, the source filename, the source file line number, and the process ID.

You can use the pseudo debug flags shown in Table 2–6 to specify which values to include.

Table 2–6 Pseudo Debug Display Flags

Pseudo Debug Flag	Description
display:name	Displays the program name registered with the cmnErrorSetProgram() function, which is off by default
display:date	Displays the date in YYYY/MM/DD HH:MM:SS format
display:file	Displays the source filename
display:line	Displays the source file line number
display:pid	Displays the process ID
display:section	Displays the debug section flag name
display:all	Displays all of the above

You can prefix the display value with '-' to turn the value off. The following example keeps the process ID from being displayed by myApp.

```
DEBUG=display:-pid
./myApp
```

Logging Alarms

The SDK provides the functions shown in Table 2–7 to use in conjunction with logging alarms.

Table 2–7 Logging Alarm Functions

Function	Description
cmnErrorSetProgram(const char * name)	Registers the program name to use when logging alarms.
const char* cmnErrorGetProgram()	Returns the name of the program previously registered by cmnErrorSetProgram() or `cmnError` if no name has been registered.
int cmnAlarmSDK (int severity, unsigned char appID, unsigned alarmTypeID, const char *const format ...)	Logs an error with alarmTypeID using variable parameter format.

The cmnAlarmSDK() function prefixes the error text with the value of `alarmTypeID`, which is associated with a specific error. The `alarmTypeID` parameter is the last four digits of the full alarm type ID , the value of which is in the range allocated for SDK applications, and is generated based on the application ID (`appId`) and the `alarmTypeID` parameter. It is in the form `<SMS Application ID><alarmTypeId>`. The SMS Application ID is specified by the `appId` argument, and will be 9xx where xx is the application ID. Custom applications use application IDs in the range 900-999.

For example, the following call to cmnAlarmSDK():

```
cmnAlarmSDK(LOGGED_ERROR, 7, 1234, "There was a problem");
```

produces an error message like the one in the following example.

```
ERROR: {9071234} SDK: There was a problem
```

Note that alarms generated from custom SDK components are distinguished from product alarms with the string "SDK".

The severity argument specifies the severity of the error message using one of the following enumerators: `LOGGED_NOTICE`, `LOGGED_WARNING`, `LOGGED_ERROR`, `LOGGED_CRITICAL`, `LOGGED_CLEAR`.

You must also define alarms in the SMF_ALARM_DEFN table, which is a database table that describes all the different types of alarms that can be generated by the system and gives information about the cause and what to do. The SMF_ALARM_DEFN table definition is shown in Table 2–8:

Table 2–8 SMF_ALARM_DEFN Table Definition

Column Name	Data Type	Description
ALARM_TYPE_ID	NUMBER(9)	The unique identifier for the alarm
DEFAULT_EVENT_TYPE	NUMBER(2)	Default value of EVENT_TYPE column.
DEFAULT_PROBABLE_CAUSE	NUMBER(4)	Default value of PROBABLE_CAUSE column.
DEFAULT_SEVERITY	NUMBER(1)	Default value of SEVERITY column.
DEFAULT_SPECIFIC_PROBLEM	VARCHAR2(256)	Default value of SPECIFIC_PROBLEM column.

Table 2–8 (Cont.) SMF_ALARM_DEFN Table Definition

Column Name	Data Type	Description
DEFAULT_RECOMMENDED_ACTION	VARCHAR2(1000)	Default value of RECOMMENDED_ACTION column.
DEFAULT_ADDITIONAL_TEXT	VARCHAR2(1000)	Default value of ADDITIONAL_TEXT column.
EVENT_TYPE	NUMBER(2)	Helps categorize the alarm, allowing quicker identification of the probable cause and recommended action.The Event Type is attached to alarm instances by the alarm definition and may be changed as required.
PROBABLE_CAUSE	NUMBER(4)	Displays the TMN standard probable causes. It is congruent with the association between event type and probable cause specified in the TMN recommendations (see ITU-T M.3100).
SEVERITY	NUMBER(1)	X.733 EFM severity: 0 (undefined) (only used by alarm_type_id 0 and -1) 1 Cleared 2 Indeterminate 3 Critical 4 Major 5 Minor 6 Warning
SPECIFIC_PROBLEM	VARCHAR2(256)	The specific identification of the fault.
RECOMMENDED_ACTION	VARCHAR2(1000)	The recommended action to resolve all instances of this type of alarm.
ADDITIONAL_TEXT	VARCHAR2(1000)	Any additional information about this type of alarm
PRESENT_TO_AM	NUMBER(1)	Indicates whether or not the alarm can be viewed in the SMS alarm management screens (viewable)
PRESENT_TO_AR	NUMBER(1)	Indicates whether or not the alarm can be presented to the alarm relay daemon (relayable)
AUTOCLEAR_PERIOD	NUMBER(10)	The auto clear period determines how long (in minutes) an alarm will be available in the Alarm tab, before it is automatically cleared by smsAlarmManager.

Table 2–8 (Cont.) SMF_ALARM_DEFN Table Definition

Column Name	Data Type	Description
REGULAR_EXPRESSION	VARCHAR2(512)	Regular expression for this alarm, if any.
NOTES	VARCHAR2(1000)	Any additional notes about this type of alarm.

The DEFAULT_... columns are only used when you edit an alarm definition in the screens and then press the Reset button. This resets the alarm definition back to the original values, as specified by the DEFAULT_... columns.

You can find additional information about alarms in the *Service Management System User's Guide* and the *Service Management System Technical Guide*.

Recording Statistics

To record statistics in your code, do the following:

1. Include the **smsStatsSDK.h** file in your code.
2. Call the smsRecordStats() function to record statistics for a given event.

The following example illustrates these steps:

```
#include <smsStatsSDK.h>

someFunction() {
  // Do something
  ...

  // Record the event
  smsRecordStats("APP", "NEW TRANSACTION", 1);

}
```

The smsRecordStats() function has the following parameters

- `application` is the name of the application recording the statistics
- `measurement` is the name of the measurement or statistic to increment
- `delta` is the increment that is being added to the measurement

You must first define the statistic in the SMS_STATISTICS_DEFN table, which has the columns described in Table 2–9:

Table 2–9 The SMS_STATISTICS_DEFN Table

Column Name	Description
STATISTIC_ID	A string that specifies the name for the statistic
APPLICATION_ID	The name of the application that generates the statistic
DESCRIPTION	A description of the statistic. For example, total calls to feature node ABC.
PERIOD	The measurement period in seconds. This is the interval at which the measurement will be collected.
COM	A short code name for the statistic. For example, sdkCall.

Accessing the Configuration File

The SDK enables you to access and read values from the configuration file, which defaults to **/IN/service_packages/eserv.config**.

Configuration is accessed using a Config object, which contains a hierarchical tree of ConfigValue objects.

The following example shows how to access the configuration file from the command line, using the Config::standard() function.

```
#include <cmnConfigSDK.h>

main(int argc, char **argv) {
  cmn::cfg::Config *configFile
      = cmn::cfg::Config::standard(argc, argv);

  std::auto_ptr<cmn::cfg::Value> ourSection;
  try {
    ourSection = std::auto_ptr<cmn::cfg::Value>(
        configFile->get("CustomApp.controlAgent"));
  } catch (cmn::cfg::Exception &e) {
    cmnAlarmSDK(LOGGED_ERROR, 0, 1005, "Cannot read config");
  }

  config.serviceKey = ourSection->get("serviceKey", 123);
  cmnAlarmSDK(LOGGED_NOTICE, 0, 1002, "serviceKey = %lld",
      config.serviceKey);

  ...

}
```

This shows how the configuration file is first read into a Config object using the Config::standard() function, then individual ConfigValue objects are extracted using the get(path) function.

For the discussion below, assume a sample section in the configuration file as follows:

```
customApp = {
  serviceKey = 123

  plugins = [
    "library1"
    "library2"
    "library3"
  ]

  controlAgent = {
    timeout = 10
    serviceKey = 45
  }
}
```

Once this configuration file has been read, elements of the configuration can be retrieved using the get() function, for example:

```
ConfigValue &controlAgent = config.get("customApp.controlAgent");
```

It is possible that the requested setting is not configured in the **eserv.config** file, in which case an exception will be thrown, which the code should take into account, for example:

```
try {
    ConfigValue &controlAgent = config.get("customApp.controlAgent");
  } catch (cmn::cfg::NotFound &e) {
    cmnAlarmSDK(LOGGED_ERROR, 0, 1001, "No customApp.controlAgent configuration
section");
    return false;
  }
```

Each ConfigValue is either a single configuration setting in the **eserv.config** file, such as the following example:

```
serviceKey = 123
```

or an array of values like this one:

```
plugins = [
  "library1"
  "library2"
  "library3"
]
```

or a map, such as the one in this example:

```
controlAgent = {
  timeout = 10
  serviceKey = 45
}
```

An array of values is represented as a cmn::cfg::ArrayValue. Each of the elements of the array can be accessed using the [] operator, for example:

```
ConfigValue &firstPlugin = plugins[0];
```

A map of values is represented as a cmn::cfg::MapValue. Each of the elements in the map can be accessed using the [] operator, for example:

```
ConfigValue &timeout = controlAgent["timeout"];
```

The get() function can also be used to select a particular setting from a map.

Once a ConfigValue has been retrieved, it needs to be converted to the expected type using the to<Type>() functions. Because the operator may have configured a configuration setting to be the wrong type, it is possible that an exception will be thrown when doing this conversion, so the code needs to take this into account, as shown in the following example:

```
  try {
    int timeoutValue = timeout.toInt();
  } catch (cmn::cfg::WrongType &e) {
    cmnAlarmSDK(LOGGED_ERROR, 0, 1002, "The customApp.controlAgent.timeout setting
is not an integer");
    return false;
  }
```

If a configuration setting has a default value, it is not necessary to check for its presence or to specify the conversion explicitly, for example:

```
int timeoutValue = controlAgent.get("timeout", 10);
```

Note that a `WrongType` exception will still be thrown if the setting has been configured with a value of the wrong type.

3

Creating Service Loaders

This chapter describes how to create a custom service loader using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Service Loaders

A service loader, which physically is a shared library that is loaded by slee_acs, is the part of Advanced Control Services (ACS) that is responsible for initializing the right service for a call and loading its control plan, profiles, and so on, if they are required. A service loader, however, is not required to execute a control plan; it can perform all of the service control logic itself.

A service loader also acts as a mediation layer between the inbound SLEE interface and the service. The service loader also performs the final manipulation of data that is returned to the network interface when a triggering interface sends back a network event.

NCC includes the following set of system service loaders that cannot be modified:

- Advanced Control Services (ACS)

  Allows service providers to define enhanced call interaction and is able to provide a variety of common call routing services. For more information, see *Advanced Control Services Technical Guide*.

- Virtual Private Network (VPN)

  Provides a virtual private network with user interfaces on industry-standard platforms. For more information, see *Virtual Private Network Service Technical Guide*.

- Charging Control Services (CCS)

  A pre-paid and post-paid service that allows customers greater flexibility and control over their billing methods and furnishes customers with an adaptable range of services. For more information, see *Charging Control Services Technical Guide*.

- Messaging Manager Service (XMS)

  A service library that specializes in handling short messages.

- Subscriber Event Service (SES)

  Enables service providers to send text messages to roaming subscribers when they roam in and out of their networks. For more information, see the *Subscriber Event Service User's and Technical Guide*.

Although you cannot modify these system service loaders, you can extend their functionality by defining custom service loader extenders. A service loader extender extends an existing product-based service loader by supplying one or more functions that are called after those in the product-based service loader.

Furthermore, you can create your own custom service loaders to manipulate information contained within the InitialDP according to your specific requirements, or to load a control plan according to your own business rules.

The InitialDP is the Intelligent Networking Application Part (INAP) request that a control agent, or external network element, sends to slee_acs to trigger processing for a new session or event, such as a voice call, data session, or SMS message.

Creating a Custom Service Loader

Service loaders are dynamically loaded libraries that are configured into a slee_acs instance at execution time. Use the SLEE configuration to steer a network originated session towards the service loader that has been written to deal with that network trigger.

Follow these steps to create a new service loader for slee_acs:

1. Determine the service key values in InitialDPs that will trigger the service loader. These values can come either directly from the network by way of the TCAP interface or through a control agent.
2. Configure the service key values within the SLEE configuration file in **/IN/service_packages/SLEE/etc/SLEE.cfg**.

   Note that comments at the beginning of the **SLEE.cfg** file describe how to configure its entries.
3. Assign service key values to a new service name within the SLEE configuration:

   The following line in **SLEE.cfg**, for example, maps service key 70 to a service called `SDK_SERVICE`:

   ```
   SERVICEKEY=INTEGER 70 SDK_SERVICE
   ```

4. Map the service name to the name of a specific slee_acs application name that will have the new service loader configured to link with it. For example, the following line specifies that the SDK_SERVICE is handled by the slee_acs application.

   ```
   SERVICE=SDK_SERVICE 1 slee_acs SDK_SERVICE
   ```

5. Map the service name to the name of the service loader's shared library in the file **/IN/service_packages/eserv.config**, which allows you to specify multiple functions to call.

   > **Note:** You can also use the legacy method of mapping the service name to the service loader's shared library in the **/IN/service_packages/ACS/etc/acs.conf** file. This method is still supported but it does not allow you to specify multiple functions to call.

6. From the shell that is executing the slee_acs instance, enter the following command to set the LD_LIBRARY_PATH environment variable:

   ```
   LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/IN/service_packages/SLEE/lib:/IN/service_
   packages/ACS/lib:directory
   export LD_LIBRARY_PATH
   ```

where *directory* is the directory where the custom service loader library has been installed.

7. Develop a shared library that includes the appropriate function entry points.

The following sections describe the functions that you can implement within a custom service loader. The ACS chassis invokes these functions at appropriate times during initialization and during the call flow. The subsections within each entry point function represent the processing steps that the service loader can perform at that point.

You can implement the following functions within your custom service loader for the ACS chassis to invoke:

- acsChassisInitSL()
- acsChassisLoadService()
- acsChassisPreCTR()
- acsChassisPreETC()
- acsChassisPrePOR()
- acsChassisCallTerminated()

acsChassisInitSL()

The ACS chassis invokes acsChassisInitSL() when it loads the shared library at startup time. You can use this function to initialize global variables and read configuration tables and files.

In the following example, the acsChassisInitSL() function simply displays a message, indicating the function has been called, and returns a value of `true`:

```
extern "C" u_int32 acsChassisInitSL() {
  DOUT << "sdkService: acsChassisInitSL(): Initial method called by service load
er" << std::endl;

  return true;
}
```

acsChassisLoadService()

The ACS chassis invokes the acsChassisLoadService() function at the beginning of a new session, call, or event, that the network starts. You can use this function to perform the following tasks in your service loader:

- Perform processing that is specific to the service name, which is set in the SLEE configuration.

  For an example of the acsChassisLoadService() function, see the file, **$NCC_SDK_HOME/example/sdkServiceLoader/example1/sdkServiceExample1.cc**

- Normalize numbers.

  The numbers that you receive from the network interface, contained within the InitialDP, are not normalized; they are identified by a Nature of Address (NoA) field and a digits field. During control plan processing, numbers are normalized; only the digits are used so these must contain sufficient information to fully identify the number. Usually this will be an E.164 number, which is generally an international telephone number.

Normalization and denormalization rules can be configured per service name within the ACS configuration file. For information on configuring these rules, see the *Advanced Control Services Technical Guide*.

The following example shows how to extract a number and normalize it:

```
char inCalled[DNODE_TN_MAX_SIZE];
char outCalled[DNODE_TN_MAX_SIZE];
u_int16 inNoA, outNoA;
ctd_getDigits(callInfo, ctd_called_num, inCalled);
inNoA = ctd_getNoA(callInfo, ctd_called_num);
if (acsNOANormalise(inNoA, inCalled, &outNoA, outCalled, DNODE_TN_MAX_SIZE)) {
     // Store the normalised number in the Normalised Called Number buffer
     ctd_setDigits(callInfo, ctd_normCalled_num, outCalled);
     ctd_setNoA(callInfo, ctd_normCalled_num, outNoA);
} else {
     cmnAlarmSDK(LOGGED_ERROR, 0, 2002, "Unable to normalise called number" );
}
```

You must decide which number fields your service will normalize and use the same principle for each one, or omit normalization altogether.

- Set up service-specific data.

  Depending on the operation, you might want to store service-specific data in an ACS memory area and access it later when this or another service loader is invoked again.

  Follow these steps to store service-specific data in an ACS memory area:

  1. Define a service-specific structure. The following code provides an example:

     ```
     class CommonServiceData : public ncc::slee::ServiceDataAPI {
     public:
              CommonServiceData();

              u_int32 status1;
              u_int32 status2;
              char data[100];
     };
     ```

  2. Call the storeServiceDataAPI() function to register an instance of the object as shown here:

     ```
     CommonServiceData *commonData = new CommonServiceData;
     ncc::slee::storeServiceDataAPI(commonData);
     ```

  3. Update the object as required. The following code provides an example:

     ```
     customSLInfo->status1 = 1;
     customSLInfo->status2 = 2;
     strncpy(customSLInfo->data, “demo”, 4);
     ```

  See the acsChassisPrePOR() function for an example of retrieving the service-specific data using retrieveServiceDataAPI().

- Check whether an emergency number is being dialed.

  Use the acsChassisIsEmergencyNumber() function to check whether a dialed number is an emergency number, as shown in the following example:

  ```
  char toCheckForEN[DNODE_TN_MAX_SIZE];
  ctd_getDigits(callInfo, ctd_called_num, toCheckForEN);
  if (acsChassisIsEmergencyNumber(toCheckForEN)) {
  ```

```
   // Handle emergency number
}
```

- Extract values from extension fields in the InitialDP.

  In some cases, you need to extract the extension parameters from the extension field of the InitialDP to use in subsequent processing or to store for later use by another component, such as a feature node. You can use the acsChassisGetGenericExtension() function to retrieve the value of a given extension field from the InitialDP.

  For an example of the acsChassisGetGenericExtension() function, see the file **sdkServiceExample1.cc** in the following location:

  $NCC_SDK_HOME/example/sdkServiceLoader/example1

- Set up profile fields that feature nodes can use or share.

  A profile is an area of memory used to store data that is derived or retrieved by a service loader.

  ACS provides temporary storage that you can use in a custom service loader to store information for use in a control plan.You can use ACS temporary-storage profile blocks to store in memory the subscriber information, or any other information, that you retrieve from the database.

  The sample code below shows how to set up such a profile block:

  ```
  setProfileField(0x09040101, myString);
  setProfileField(0x09040102, myInteger);
  setProfileField(0x09040103, (const void*)&myCustomData, sizeof(myCustomData));
  ```

  You must allocate profile tags for custom applications as hexadecimal numbers in the range 0x0aaa0000 to 0x0aaaFFFF where aaa is the hexadecimal representation of the application ID. You can choose custom application IDs from the range 900 - 999, which is 0x384 - 0x3E7 in hexadecimal.

  For more information, see the setProfileField() function in the **acsChassisProfileSDK.h** file in the API reference documentation.

  > **Note:** The master database for profiles is only updated at session end and not in mid-session. This is a built-in performance feature. Therefore, if you update a tag within the control plan, be aware that the tag will not be updated and replicated in the database in mid-session.

  You must determine the profile tag, which is the first parameter passed to setProfileField(), and it must not conflict with any other profile tags that you need to create. Oracle recommends that you define the profile tag within a header file that is available to both the custom service loader and any custom feature node that might need to retrieve the data from the profile. You also need to configure these profile tags in the ACS configuration screens to make them available to control plans.

- Load the control plan based on name.

  The acsGetControlPlan() function allows you to load a control plan to be executed by ACS. For example:

  ```
  std::string controlPlanName = "MyControlPlan";
  if (not acsGetControlPlan(controlPlanName.c_str(), dbData, callInfo)) {
  ```

```
        // Set some release cause chosen to indicate "no control plan"
        con_setReleaseCause(outData, 1);
        return RELEASE;
    }
```

acsChassisPreCTR() and acsChassisPreETC()

The acsChassisPreCTR() function enables you to control FurnishChargingInformation (FCI) and SendChargingInformation (SCI) that is sent with outbound Connect To Resource (CTR) or ReleaseCall operations. The acsChassisPreETC() function enables you to control FurnishChargingInformation and SendChargingInformation that is sent with outbound EstablishTemporaryConnect (ETC) operations.

FurnishChargingInformation and SendChargingInformation are INAP operations that you can send from the SLC to the service switching point (SSP). You define the content of the data depending on the customer's requirements. It is defined as an operator-specific octet string.

The FurnishChargingInformation operation allows you to control call data records that the SSP produces. One example could be a customer who has a custom service loader that requires adding a list of played announcements and the number of connection attempts made to the AMA record that the SSP produces.

The SendChargingInformation operation mainly influences how much the SSP charges, if the SSP is doing the charging. For example, for a customer who has a friends and family service, SLC responds to the InitialDP operation for a normal call with Continue, but for a friends and family call, it responds with SendChargingInformation.

You can also call the acsChassisPrePOR() function to send FCI and SCI. See "acsChassisPrePOR()" for more information.

acsChassisPreCTR()

The acsChassisPreCTR() function is defined in the **service-loader.h** file and allows you to specify the FCI contents in the outgoingCallInfo_t structure.

The function is called by slee_acs in the following way before sending CTR operations:

1. The slee_acs process calls acsChassisPreCTR() when it needs to send a Connect or ReleaseCall operation.
2. The acsChassisPreCTR() function calls one or both of the following sets of functions:
   - The con_setDoWeFCI() and con_setFCI() functions to specify FCI contents
   - The con_setDoWeSCI() and con_setSCI() functions to specify SCI contents
3. When the acsChassisPreCTR() function returns, the slee_acs process sends either the FCI or SCI operation or both. Then slee_acs sends the CTR operation.

You declare the acsChassisPreCTR() function in the following way:

```
extern "C" void acsChassisPreCTR (
     callTelephonyData_t *callInfo,
     outgoingCallInfo_t *outData,
     acsEngineContext *engineFields);
```

The parameters have the following definitions:

- The **callInfo** parameter stores chassis information about the call, which is derived from the IDP. The **callinfo** parameter is a pointer to an instance of the

callTelephonyData_t class. You can obtain access to this information through the callTelephonyData_t getter and setter functions.

- The **outData** parameter sets the information to return to the network. The **outData** parameter is a pointer to an instance of the outgoingCallInfo_t class. You can obtain access to this information through the outgoingCallInfo_t getter and setter functions.
- The **engineFields** parameter stores information about the state of an individual call while it is being processed by the engine.

acsChassisPreETC()

The acsChassisPreETC() function is defined in the **service-loader.h** file and allows you to specify the FCI contents in the outgoingCallInfo_t structure.

The function is called by slee_acs in the following way before sending ETC operations:

1. The slee_acs process calls acsChassisPreETC() when it needs to send an EstablishTemporaryConnect operation.
2. The acsChassisPreETC() function calls one or both of the following sets of functions:
   - The con_setDoWeFCI() and con_setFCI() functions to specify FCI contents
   - The con_setDoWeSCI() and con_setSCI() functions to specify SCI contents
3. When the acsChassisPreETC() function returns, the slee_acs process sends either the FCI or SCI operation or both. Then slee_acs sends the ETC operation.

You declare the acsChassisPreETC() function in the following way:

```
extern "C" void acsChassisPreETC (
     callTelephonyData_t *callInfo,
     outgoingCallInfo_t *outData,
     acsEngineContext *engineFields);
```

The parameters have the following definitions:

- The **callInfo** parameter stores chassis information about the call, which is derived from the IDP. The **callinfo** parameter is a pointer to an instance of the callTelephonyData_t class. You can obtain access to this information through the callTelephonyData_t getter and setter functions.
- The **outData** parameter sets the information to return to the network. The **outData** parameter is a pointer to an instance of the outgoingCallInfo_t class. You can obtain access to this information through the outgoingCallInfo_t getter and setter functions.
- The **engineFields** parameter stores information about the state of an individual call while it is being processed by the engine.

acsChassisPrePOR()

The prePOR part of this function's name stands for pre-point-of-return, indicating that it allows the service loader to do something at the last minute, such as denormalize a number, or add an additional bit of information to the outgoing message or event.

ACS calls acsChassisPrePOR() when a feature node within the control plan requests a specific network action or when acsChassisLoadService() returns a response that causes a specific network action.

You can use the con_getPORToAttempt(outData) function to see which type of INAP/TCAP action the feature node sends to the network.

You can override the control plan request by setting the acsChassisPrePOR() function's return value. See acsPrePostProcessingReturn_t in the acsServiceEntryAccessSDK.h file in the SDK API reference documentation for possible return values.

For an example of the acsChassisPrePOR() function, see the file **$NCC_SDK_HOME/example/sdkServiceLoader/example1/sdkServiceExample1.cc**.

The following sections describe some of the tasks that you might perform in the acsChassisPrePOR() function.

Denormalization

If a CONNECT request is to be sent back to the network, it is standard practice to first denormalize the numbers in the operation. This does not have to be done if other feature nodes have previously denormalized the numbers or if the service loader performs service-specific processing on the numbers.

Denormalization is usually required because the ACS control plan engine expects numbers in normalized form, but INAP operations expect numbers in denormalized form; that is, NoA and digits. Typically, a normalized number would be in the full E.164 international format while a denormalized number might be in the form it's used within an area code. For example, 447880555555 would be the normalized format of a United Kingdom phone number, while an NoA value of 2 (unknown) and the digits 07880555555 could be the denormalized version of the number.

You can denormalize numbers by using functions on the outData argument that acsChassisPrePOR() receives, as seen in the following example:

```
// Denormalise Destination Routing Address
 char inDRA[DNODE_TN_MAX_SIZE];
 char outDRA[DNODE_TN_MAX_SIZE];
 u_int16 outNoA;
 con_getDigits(outData, con_normDRA_num, inDRA);
 acsNOADenormalise(acsNOANotApplicable(),
                inDRA,
                &outNoA,
                outDRA,
                DNODE_TN_MAX_SIZE);
 con_setDigits(outData, con_DRA_num, outDRA);
 if (outNoA != acsNOANotApplicable()) {
     con_setNoA(outData, con_DRA_num, outNoA);
 }
```

Setting up Service-Specific Data

In the acsChassisPrePOR() function you can also examine or access data that was previously set up by the acsChassisLoadService() function. Likely you will access rather than store service-specific data because you will most likely populate a custom field or extension in the outgoing request based on data that you have already set up.

The following code shows how to examine and modify data using the service-specific type `CommonServiceData` as shown in the acsChassisLoadService() example. The code sets the `someField` element, which `commonData` points to, to a value of 99.

```
CommonServiceData *commonData =
     dynamic_cast<CommonServiceData*>(ncc::slee::retrieveServiceDataAPI());
if (commonData) {
     commonData->someField = 99;
}
```

Setting up Extension Information

Typically, the information you populate an extension with corresponds with information you got from service-specific data, or from temporary storage or profile tags that the service loader or a feature node previously populated.

For some network interfaces, the service loader can use the extension field that is sent as part of the INAP or TCAP CONNECT operation to transport information that the interface can use to govern functionality or to send information back to the network in a protocol-specific form. Similar to retrieving information within the InitialDP, to use this functionality, you must know the tag values and types that the interface uses. You can obtain these tag values and types in the acsChassisLoadService() function.

The following example sets up an extension field to contain a string value of "50" for a parameter with the tag value of 100.

```
// Find an empty extension
for (int32 j = 0; j != MAX_OUTGOING_EXTENSIONS; ++j) {
     int bLength = 2048;
     unsigned char buffer[bLength];
     unsigned int tag = 0;
     if (acsChassisGetGenericExtension(outData, j, tag, buffer, bLength)) {
         if (tag == 0) {
             // ... and write the new extension to it
             acsChassisSetGenericExtension(outData, j, 100, "50");
         }
     }
}
```

General Setup of Outgoing Information

You can set up the cause of a release operation by calling the con_setReleaseCause() function.

You can setup the destination routing address for a CONNECT operation by using the con_setDigits() and con_setNoA() functions as shown in the following example:

```
con_setDigits(outData, con_normDRA_num, "12345678");
con_setNoA(outData, con_normDRA_num, 3);
```

You can set up cut-and-paste parameters using the con_setCutAndPasteFlag() and con_setCutAndPasteNumDigits() functions.

Sending FurnishChargingInformation or SendChargingInformation

You can call con_setDoWeFCI() and con_setFCI() functions to specify FurnishChargingInformation or call con_setDoWeSCI() and con_setSCI() functions to specify SendChargingInformation contents. If so, the FurnishChargingInformation or SendChargingInformation, or both, is sent before the Connect operation.

See "acsChassisPreCTR() and acsChassisPreETC()" for more information on FurnishChargingInformation and SendChargingInformation.

acsChassisCallTerminated()

You can call acsChassisCallTerminated() from within a service loader to perform post-call cleanup when a call has been terminated.

Any service-specific data that was created within the acsChassisLoadService() function is available for you to use, for example, to generate a custom Event Detail Record.

For an example of the acsChassisCallTerminated() function, see the following file:

$NCC_SDK_HOME/example/sdkServiceLoader/example1/sdkServiceExample1.cc

Defining a Custom Service Loader Extender

You can use the example service loaders as service loader extenders by configuring a service as follows in the **eserv.config** file:

```
ACS = {
    ServiceEntries = [
        {
        AddressSources = { }
        MinSleeEventSize = 2048
        ServiceName = "SDK_SERVICE"
        Methods = {
            acsChassisCallTerminated = [
                "libsdkServiceExample1.so"
            ]
            acsChassisPrePOR = [
                "libsdkServiceExample1.so"
            ]
            acsChassisInitSL = [
                "ccsSvcLibrary.so"
                "libsdkServiceExample1.so"
            ]
            acsChassisLoadService = [
                "ccsSvcLibrary.so"
                "libsdkServiceExample1.so"
            ]
        }
        }
    ]
}
```

These entries cause ACS to do the following:

- Run the standard CCS service loader's acsChassisLoadService() function, which does the standard CCS product processing, such as looking up a subscriber, loading the subscriber profile, determining a control plan, and so on
- Run the example service loader's acsChassisLoadService() function, which gets some extension information out of the received InitialDP and then overrides the control plan

The example service loader, **libsdkServiceExample1.so**, is available in the SDK in the **$NCC_SDK_HOME/lib** directory.

4

Creating a Custom Feature Node

This chapter describes how to create custom feature nodes using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Feature Nodes

A feature node represents a unit of functional logic that you can include within an NCC control plan to implement a portion of the plan's overall logic flow.

The Advanced Control Services (ACS) component of NCC includes a set of default feature nodes that you can access to build a control plan within the NCC Control Plan Editor.

Other NCC components such as Prepaid Charging, Messaging Manager, and Number Manager also include feature nodes that you use to perform specific operations. These operations include charging for a data session, sending an SMS or USSD message, and performing a database lookup, for example.

The following are some NCC components that include feature nodes. You cannot modify these system feature nodes, but you can use them to build a control plan:

- Subscriber Event Service (SES) enables service providers, or network operators, to send text messages to roaming subscribers when they roam in or out of their network.
- Data Access Pack (DAP) provides the ability to send requests to external application service providers (ASPs) and receive responses for further processing by the Service Logic Controller.
- Open Services Development (OSD) enables NCC to provide control plans as a web service. NCC supports generation of WSDL (web services description language) files automatically based on control plans. NCC provides the WSDL to third parties such that the third party platform knows how to generate and invoke a web services operation and, consequently, a control plan on the NCC system.
- Advanced Control Services (ACS) allows service providers to define enhanced call interaction that is triggered when one or more of the following calls occur:
  - Calls to specific dialed numbers (Service Numbers)
  - Calls from specific calling numbers (CLI numbers)
  - All calls triggered to a specified INAP service key
- Virtual Private Network (VPN) connects multiple locations with each network having its own private numbering plan or mapping numbers in the private plan to the numbers required to correctly route the call via the public switched telephone

network (PSTN) or mobile network. Additional processing can be done to further add value to the service

- Charging Control Services (CCS), which handles online charging
- Messaging Manager (XMS)
- Location Capability Pack (LCP), which provides location services that identify where the mobile subscriber is
- Unstructured Supplementary Services Gateway (USSD GW), which is a control agent that handles a mobile-initiated USSD conversation. For example, when a mobile subscriber enters *200# on the phone and receives a menu and then enters a number to select a menu option, the USSD Gateway handles this interaction. It sends INAP requests to slee_acs to trigger control plan processing, which defines the service logic.

You can also create a custom feature node to implement functional logic that is specific to your business requirements.

About Creating Custom Feature Nodes

The process of creating a custom feature node consists of the following steps:

1. Defining the feature node and loading the definition into database tables.
2. Adding it to an ACS feature node set.
3. Creating the shared library.
4. Specifying the location of the shared library.
5. Creating the feature node image files.

Defining a Feature Node

Before you can use a feature node within the Control Plan Editor, you must first define it and the parameters and functional branches that it offers.

You define your feature node in an XML file and load the definitions into the Service Management System (SMS) database by running the **acsMacroNodeInstaller** utility.

Creating a Feature Node Definition

To define a feature node you create an XML file using the XML elements that Table 4–1 describes:

Table 4–1 Feature Node Definition Elements

XML Element	Description
<MacroNodeDefinitions>	The top-level element that contains all other elements.
<Nodes>	Contains a <Node> element for each feature node you want to define.
<Node>	Contains the elements that define a specific feature node.
<Name>	Specify a unique name for the feature node, which is stored internally.
<FastKey>	Specify the three character fast key identifier associated with this node

Table 4–1 (Cont.) Feature Node Definition Elements

XML Element	Description
<DefaultNodeName>	Optional. Specify a default name for the node, such as an abbreviated version of the value for <Name>. Defaults to the value of the <Name> element, if not specified.
<DisplayName>	Specify a user-friendly name, which can include spaces, that is displayed in the Control Panel Editor in the Advanced Control Services UI.
<Permission>	The permission level required for a user to use this feature node. Users and their permission levels are defined on the **Users** tab of the ACS Customer screen in the ACS UI.
<Description>	A description of the feature node
<Group>	The name of the feature palette to which the feature node belongs. The feature palette is the area of the Control Plan Editor in the ACS UI that displays the feature node sets that are available to the user.
<NodeVersion>	Numeric version number, which allows multiple versions to be supported. For example, if you add a parameter to the feature node and assign the node version number 2, the code should check the version number and substitute a default value for the new parameter if it's an old version of the node.
<UsesTelephony>	Specifies whether the node does any telephony operations. More accurately, it specifies whether it requires that the main dialog is still active. Value can be true or false and is case sensitive. Defaults to false. The Control Plan Editor imposes the rule that you cannot connect an exit branch with TelephonyAllowed=false to a subsequent feature node that has UsesTelephony=true.
<Parameters>	Contains the <Parameter> elements for each parameter you want to define.
<Parameter>	Contains the <Name>, <Group>, <Type>, and <DefaultData> elements that define a specific parameter.
<Name>	The label attached to the parameter in the node GUI
<Group>	The label or name of the parameter group in the GUI to which this parameter belongs
<Type>	Values: radiobutton; table; checkbox; integerfield; stringfield; profilefield.
<DefaultData>	The default data to assign to the parameter, if none is specified.
<ExitBranch>	Contains the <Name> and <TelephonyAllowed> elements that define a specific exit branch for the feature node.
<Name>	Name of the exit branch.
<TelephonyAllowed>	Specifies whether the next feature node called can be one that performs a telephony action. A value of true allows a telephony action; false does not. The Control Plan Editor does not allow you to follow a "no telephony" exit branch with a node that implements a telephony action. For example, if you add a branch that indicates the caller has hung up, you would specify false for TelephonyAllowed. After that, you can't use a node that performs telephony because the call has ended, but you can still use nodes to complete the charging or send a notification.

Example: Feature Node Definition File

The following statements define a feature node named ExampleAttemptTerminate:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE MacroNodeDefinitions SYSTEM "nodeschema.dtd"[]>

<MacroNodeDefinitions>
<Nodes>
  <Node>
    <Name>ExampleAttemptTerminate</Name>
    <FastKey>EAT</FastKey>
    <DefaultNodeName>ExampleAT</DefaultNodeName>
    <DisplayName>Example Attempt Terminate</DisplayName>
    <Permission>1</Permission>
    <Description>Attempt to terminate call to either dialled number or configured
parameter</Description>
    <Group>Example</Group>
    <NodeVersion>1</NodeVersion>
    <Parameters>
        <Parameter>
            <Name>Number</Name>
            <Group>Number</Group>
            <Type>stringfield</Type>
            <DefaultData></DefaultData>
        </Parameter>
    </Parameters>
    <ExitBranches>
     <ExitBranch><Name>Success</Name><TelephonyAllowed>false</TelephonyAllowed></E
xitBranch>
     <ExitBranch><Name>Busy</Name><TelephonyAllowed>true</TelephonyAllowed></ExitB
ranch>
     <ExitBranch><Name>RouteSelectFailure</Name><TelephonyAllowed>true</TelephonyA
llowed></ExitBranch>
     <ExitBranch><Name>Abort</Name><TelephonyAllowed>false</TelephonyAllowed></Exi
tBranch>
     <ExitBranch><Name>Abandoned</Name><TelephonyAllowed>false</TelephonyAllowed><
/ExitBranch>
     <ExitBranch><Name>NoAnswer</Name><TelephonyAllowed>true</TelephonyAllowed>
     </ExitBranch>
     <ExitBranch><Name>DisconnectLeg1</Name><TelephonyAllowed>false</TelephonyAllo
wed>
</ExitBranch>
     <ExitBranch><Name>DisconnectLeg2</Name><TelephonyAllowed>false</TelephonyAllo
wed></ExitBranch>
     <ExitBranch><Name>Error</Name><TelephonyAllowed>false</TelephonyAllowed></Exi
tBranch>
    </ExitBranches>
  </Node>
```

Loading Feature Node Definitions

You load feature node definitions into the database by running the **acsMacroNodeInstaller** utility. The **acsMacroNodeInstaller** command-line utility reads a set of feature node definition parameters from the XML file that you created and loads the definitions into the SMS database.

The utility can create and delete feature node definitions, and update the definitions of existing feature nodes.

Run the **acsMacroNodeInstaller** utility by using the following syntax:

```
acsMacroNodeInstaller {-install | -update | -uninstall}
                      [-D destination_file] [-v] [-h | ?] XML_file
```

Where:

- -**install** installs the feature nodes defined in the XML file
- -**update** updates the feature nodes defined in the XML file
- -**uninstall** uninstalls the feature nodes defined in the XML file
- **-D** *destination_file* is the path and name of the debug file; for SQL only
- **-v** verbose mode
- **-h** or **?** shows help for this command
- *XML_file* is the path and name of the XML feature node definition file that you created.

Adding the Feature Node to a Feature Set

For the procedure to add a feature node to a feature set, see the discussion on ACS configuration in *Advanced Control Services User's Guide*. Only the ACS system administrator can access the ACS Feature Sets screen.

Creating the Shared Library

You implement the logic of the custom feature node in a C++ shared library. The logic of the feature node is implemented within a framework that enforces certain predictable interactions with ACS and the Control Plan Editor. This framework is constructed through the use of particular API functions. From a functional standpoint, the framework consists of the following tasks:

- Initialization, in which the shared library registers any nodes that can be called within it
- Processing, which is managed through the implementation of a processor function that is of type `acsEngineFNProcessor`.

  This function can be called multiple times during control plan processing. The multiple entries are typically handled by implementing what's called a state machine. A state machine is an abstract model in which a computer program records its current state as an integer and later uses the value of the integer to branch to a particular function to perform the processing that's required at that point. For an example of this type of processing, see "Tracking the State".
- Exit branches

  A feature node normally exits through one of its branches so the next node in the control plan can begin executing.

Initialization

All feature nodes must be part of a shared library that is configured to be loaded by slee_acs. Initialization occurs when slee_acs starts up and calls the shared library's `sharedLibraryInit()` function, which you implement to register the feature nodes that the shared library contains. You register a feature node by calling the `acsEngineRegisterFNType()` function. This function has two parameters, the first identifies the node being registered by its fast key value, and the second specifies the

name of the callback function that is the node's entry point when it's invoked by the control plan and also when a requested action has been completed.

If the node uses a node context, you must call the `acsEngineRegisterFNTypeSize()` function to register its size. For more information about a node context, see "Using the Node Context Block".

The following example, which you can find in the API reference documentation (see **$NCC_SDK_HOME/doc/html/index.html**) implements the `sharedLibraryInit()` function, which registers a feature node that has a fast key value of `_NOD` and a callback function named `processFunction()`. Note that the underscore prefix for the fast key is required. The `sharedLibraryInit()` function also calls `acsEngineRegisterFNTypeSize()`to register the size of the node context. The registration functions are called inside `if` statements to determine whether registration occurred successfully. The `sharedLibraryInit()` function returns a value of 1 on successful completion. Otherwise, it returns 0

```
extern "C" u_int32 sharedLibraryInit() {
    // Register first node
    if (not acsEngineRegisterFNType(acsEngineFastKey("_NOD"), processFunction)) {
        return 0;
    }
    // Register the node's context data size
    if (not acsEngineRegisterFNTypeSize(acsEngineFastKey("_NOD"),
sizeof(NodeContext))) {
        return 0;
    }

    // Register other nodes
    // ...

    return 1;
}
```

Processing

The callback function that a feature node registers against its fast key value must be of type `acsEngineFNProcessor`. The ACS engine calls this function each time it determines that the feature node has processing to perform. The following steps summarize a typical interaction between the ACS engine and the feature node's callback function:

1. The ACS engine enters the node for the first time, calling the node's processing function.
2. The node requires an external action, sets up the action, and returns from the processing function with the `ACS_ENGINE_MACRO_STAY_HERE` result.
3. The ACS chassis executes the action and passes control back to the node by calling the process function again with the result of the action.
4. The node continues processing, performing more actions in the same way until it finishes processing.
5. The node specifies an exit branch and returns `ACS_ENGINE_MACRO_FOLLOW_BRANCH`.

Note: As slee_acs processes only one call at a time, all other calls must wait while a node's process() function is running. You must ensure, therefore, that you do not perform any blocking actions or excessive calculations that would unnecessarily delay returning execution to the ACS engine. Instead, you should perform such processing by invoking a chassis action. For more information, see "Making a Chassis Action Request".

Tracking the State

During interaction with the ACS engine, the feature node must keep track of its current state. You can accomplish this by implementing a *state machine*, which is a mechanism that stores the state of an operation at a particular point in time in an event-driven workflow.

To implement a state machine, use the `acsEngineContext` structure, defined in `acsEngineSDK.h`, to store the node's current state. Use `aec_setNodeState()` and `aec_getNodeState()` to access the state.

The following example illustrates how the main processing callback function can access `acsEngineContext` to determine the node's current state.

```
// State functions
acsEngineFNProcessor nodeProcessStep1;
acsEngineFNProcessor nodeProcessStep2;
acsEngineFNProcessor nodeProcessStep3;

acsEngineFNProcessor *nodeStateFunctions [] = {
  nodeProcessStep1,
  nodeProcessStep2,
  nodeProcessStep3
};

u_int32 mainNodeProcessFunction(acsEngineContext *context,
     acsChassisActionGenericResult *result,
     acsChassisAction *action,
     void *nodeContext,
     u_int32 *branch) {

  // Lookup and call the function appropriate to the current state
  return nodeStateFunctions[aec_getNodeState(context)](context,
     result,
     action,
     nodeContext,
     branch);
}
```

Here the `return` statement looks up the function appropriate to the current state in the `nodeStateFunctions` array. The return statement could also be divided into three separate statements as shown in the following example:

```
acsEngineFNProcessor *stateFn = nodeStateFunctions[aec_getNodeState(context)];
u_int32 processResult = (*stateFn)(context, result, action, nodeContext, branch);
return processResult;
```

The following example shows the type of processing that would be done by one of the functions that's called by the main processing callback function.

```
u_int32 nodeProcessStep1(acsEngineContext *context,
```

```
    acsChassisActionGenericResult *result,
    acsChassisAction *action,
    void *nodeContext,
    u_int32 *branch) {

  // Do some processing and set up an external action
  // ...

  aec_setNodeState(context, acsEngineState2);
  return ACS_ENGINE_MACRO_STAY_HERE;
}
```

The last two lines here set the next processing state and return a result that indicates that a chassis action is required. The processing state determines which function will be called when control is returned to the node after the external action.

Making a Chassis Action Request

When a feature node performs an external action, it requests it from the ACS Chassis and suspends execution until the Chassis returns the result. Table 4–2 lists the actions that a node can request from the Chassis and also lists the files where the actions are defined:

Table 4–2 ACS Chassis Actions

Actions	Files
Charging	`acsActionsSDK.hh`
Telephony	`acsChassisActionSDK.h` for requests `acsChassisStatusSDK.h` for responses
Notification	`acsNotificationProcessorSDK.h`
Replication	`acsChassisActionSDK.h`
Call-out	`GenericSendActionHandlerSDK.h`

To invoke an action, the node sets up the request in the argument of the action's function and returns a value of `ACS_ENGINE_MACRO_STAY_HERE`.

The Chassis executes the requested action and then calls the node's callback function with the result.

For example, the node might want to request a retrieve-profile action and then move to the next state to receive the result. In the following example, the code stores the state that it has reached and then returns control to the part of ACS that called the process function. Once ACS completes the chassis action, it calls the node's main process function again, which looks at the current state to determine which function among `nodeStateFunctions` to call.

```
// ...

  doRetrieveProfileAction(action, context);
  addRetrieveProfileTag(action, PROFILE_TEMPORARY_STORAGE, 0x3E90001);

  aec_setNodeState(context, acsEngineState2);
  return ACS_ENGINE_MACRO_STAY_HERE;
}
```

When the callback function, `mainNodeProcessFunction()`, is called for the second time, it calls `nodeProcessState2()`, which interprets the result of the action using the `result` argument.

```
u_int32 nodeProcessStep2(acsEngineContext *context,
     acsChassisActionGenericResult *result,
     acsChassisAction *action,
     void *nodeContext,
     u_int32 *branch) {

    if (getRetrieveProfileResultValid(result)) {
        acsProfileField *fieldPtr = aec_getProfileField(context, 0);
        if (fieldPtr) {
           std::string fieldValue;
           if (cmnBufferGetValue(fieldPtr->buffer, fieldPtr->length, fieldValue)){
                // Do something with the retrieved field
           }
        }
    }
```

Exiting

When the feature node finishes its operations, it exits and returns control to the control plan by taking one of its defined branches.

The following example illustrates how to do this, setting `branch` returning the constant `ACS_ENGINE_MACRO_FOLLOW_BRANCH`:

```
#define NODE_BRANCH_SUCCESS  0
#define NODE_BRANCH_FAILURE  1

u_int32 nodeProcessStep3(acsEngineContext *context,
     acsChassisActionGenericResult *result,
     acsChassisAction *action,
     void *nodeContext,
     u_int32 *branch) {

     // Determine the outcome of the node processing
     // ...

     *branch = NODE_BRANCH_SUCCESS;
     return ACS_ENGINE_MACRO_FOLLOW_BRANCH;
}
```

Using the Node Context Block

At initialization time, the feature node can request the ACS engine to allocate a block of memory that the node can use to keep track of its context between calls. The node requests the block of memory and specifies its size by calling the `acsEngineRegisterFNTypeSize()` function in `sharedLibraryInit()`.

The ACS engine manages the context memory block so the node does not need to free it when it has finished with it. The node does need to cast the memory block to the type that it requires, however.

In the following example, the node defines the context memory as a `struct` that is made up of an integer and an array of ten characters. It then calls `acsEngineRegisterFNTypeSize()` in `sharedLibraryInit()` to ask the ACS engine to allocate the context block as `sizeof(NodeContext)`.

```
struct NodeContext {
```

```
    int field1;
    char field2[10];
};

extern "C" u_int32 sharedLibraryInit () {
    ...
    // Register the node's context data size
    if (not acsEngineRegisterFNTypeSize(acsEngineFastKey("_NOD"),
sizeof(NodeContext))) {
        return 0;
    }
    ...
}
```

In the first processing function, `nodeProcessStep1()` defines `context` as a pointer of type `NodeContext` and sets it to the value of `nodeContext`, a function parameter that points to the node's context memory block. It then defines two local variables, `std::string param2` and `paramOffset`, which it uses to access the block's content. The `getMacroNodeParameter()` function retrieves data from the block at the offset specified by `paramOffset` into the location specified by the second parameter. The first call loads `field1` of `NodeContext`, while the second call and the following call to `strlcpy()` load `field2`. Note that the first call to `getMacroNodeParameter()` also increments `paramOffset` by the length of `field1`.

```
u_int32 nodeProcessStep1(acsEngineContext *context,
     acsChassisActionGenericResult *result,
     acsChassisAction *action,
     void *nodeContext,
     u_int32 *branch) {

  NodeContext *context = (NodeContext *) nodeContext;

  std::string param2;
  u_int32 paramOffset = 0;
  getMacroNodeParameter(paramOffset, context->field1);
  getMacroNodeParameter(paramOffset, param2);
  strlcpy(context->field2, param2.c_str());

  ...
}
```

Specifying the Location of the Shared Library

After creating the shared library that contains the feature node's functional logic, you must update the LD_LIBRARY_PATH environment variable to identify its location. Usually, you start slee_acs using the following script or one like it:

/IN/service_packages/ACS/bin/slee_acs.sh

To add the location of a custom feature node library, modify the script to add the library's location to `LD_LIBRARY_PATH`, as shown in the following example:

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH:}:<path to shared library>MyMacroNode.so
```

Creating the Feature Node Image Files

Feature nodes are represented in the Control Panel Editor by a graphic image. You must create a graphic image to represent your new feature node and add it to the

image folder where the Control Panel Editor can access it. You can use the graphic tool of your choice to create the image.

To create the graphic image and make it available to the Control Plan Editor:

1. Create an image that represents your feature node. The image must be 64 pixels high by 40 pixels wide. Name the image file FN*nodename*.png. The image will be used in the main Control Plan Editor panel.
2. (Optional) Create a tooltip file in HTML format that describes the feature node. Name the file TT*featureNodeType*.htm. This text will be displayed in the CPE feature node palette when the cursor hovers over the node's image.
3. Place the image file in the following location on the Service Management System (SMS):

   /IN/html/Acs_Service/images/
4. If you create a tooltip file, place it in the following location on the Service Management System (SMS):

   /IN/html/Acs_Service/helptext/Default/

5

Creating a Custom Control Agent

This chapter describes how to create a custom control agent using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Control Agents

A control agent is a process that provides a bridge between a particular protocol and the service logic being run by slee_acs. To provide the bridge, the control agent must maintain a protocol-specific dialogue with some network component on one side while handling the INAP interaction with slee_acs on the other side.

The SIP control agent (SCA) is a good example. SIP is a TCP/IP based, HTTP-like, protocol for handling multi-media session control. The SCA provides the translation layer between the SIP protocol and the native INAP protocol that the SLC supports internally. This allows the SLC to act as an SIP proxy and implement complex services for VOIP applications, such as VPN or prepaid charging.

The TCAP API provides the interface for a TCAP-based control agent to communicate with another network element.

The INAP API provides the interface that the control agent needs to trigger ACS, by sending and receiving the required INAP operations.

For TCAP-based protocols, the control agent receives a TCAP payload and uses its routines to decode the payload before triggering ACS using an API. The agent receives messages from ACS through an API and it can use these messages to build the payload to return in a TCAP response; an appropriate API retrieves the TCAP payload and sends it. You can use this to support specific versions of INAP, for example.

Assume, for example, that you want to integrate with a service switching function that does not support a standard INAP protocol because it has custom or non-standard message flows and parameters. The protocol is still encapsulated in standard TCAP. To support it you can develop a custom control agent that extracts, interprets, and translates the non-standard INAP protocol. Then you can use the NCC INAP API to communicate with ACS.

For non-TCAP-based protocols, you are responsible for writing the network facing transport or code. However, you will still use the INAP API to send and receive messages to ACS as in the case of the TCAP-based control agent.

SLEE Dispatcher

Both the INAP API and the TCAP API use the Service Logic Execution Environment (SLEE) to communicate with other processes.

The `slee::Dispatcher` class and its associated `slee::Transaction` objects handle all interaction with the SLEE. The `slee::Dispatcher` class checks for events when the Control Agent's main loop requests it and passes them to the appropriate `slee::Transaction` handler, or if necessary, creates a new transaction.

The `slee::Dispatcher` has a `poll()` function that checks for any new events and needs to be called explicitly by the custom program. Each handler is an instance of the `slee::Transaction` class and is associated with a particular SLEE dialog. A factory object can create a new transaction for a new dialog.

The `slee::TransactionFactory` interface defines a factory for creating transactions of a particular type. This factory can then be registered with the dispatcher as the factory to use for a particular event type. Whenever the dispatcher receives an event on a new dialog, it creates an inbound transaction using the associated factory.

The SDK provides the following base transaction types:

- `acs::AcsSleeTransaction`
- `tcap::TcapSleeTransaction`
- `GenericSleeTransaction`

As a control agent or call-out interface developer, you are responsible for deriving your own transaction classes from one of these transaction types. Then call the slee::Dispatcher::poll() function periodically to check for events and pass them to the associated transaction instance.

The SDK TCAP API

For a description of the TCAP protocol, see the website:

http://www.itu.int/rec/T-REC-Q.771-199706-I/en

The SDK TCAP API provides the component and transaction sub-layers of the protocol.

The `tcap::TcapSleeTransaction` object manages an incoming TCAP dialogue. To provide behavior that is specific to your application, you must implement a class that is derived from the `tcap::TcapSleeTransaction` class.

The control agent registers its transaction class using the `registerTransactionFractory()` method, as shown in the following example.

```
using namespace ncc;

slee::Dispatcher dispatcher;
slee::TransactionFactory<CustomTcapTransaction> factory;
dispatcher.registerTransactionFactory(&factory);
```

Whenever the `slee::Dispatcher` receives a `TC-BEGIN` message on a new SLEE dialogue, it creates an instance of this transaction class and passes all incoming `tcap::Primitives` to it. The transaction class can send back its own primitives using the `tcap::TcapSleeTransaction::sendTcapPrimitive()` function.

The TC-user communicates with the component sub-layer using primitives. The API models this communication. Each primitive can be a request from the TC-user telling the component sub-layer to perform some function, or it can be an indication from the component sub-layer.

For example, the control agent can use the API to send an Invoke request primitive to request an operation on the remote system. Once the remote system has performed the operation, the control agent then receives a Result indication primitive.

Alternatively, the remote system might ask the control agent to invoke an operation, in which case the control agent will receive an Invoke indication from the API and send back a Result request.

The TCAP data classes provide the developer an interface that models the interaction between a TC-user, the application that is using TCAP, and the component sub-layer defined in ITU Q.771 or ANSI T1.114.1, the European and American versions of the TCAP protocol, respectively.

For more information on ITU Q.771 see:

http://www.itu.int/rec/T-REC-Q.771-199706-I/en

Both dialogue and component-handling primitives are derived from a common `tcap::Primitive` base class, that contains the following information:

- ITU / ANSI flag (type `TcapProtocolVersion`)
- Dialogue ID

Primitives can be either requests from the TC-user or indications to the TC-user. Any parameter that is available in the request primitive has a setter and any parameter that is available in the indication primitive has a getter. Use the setter functions when constructing a request primitive. They let you set the various parameters of the primitive. Use the getter functions to extract parameters from an indication primitive that the control agent receives.

The dialogue primitives request or indicate facilities of the underlying transport layer in conjunction with controlling the dialogue or message flow. The component primitives handle operations and replies, which are the content of the dialogue. These component primitives do not require facilities from the underlying transport layer.

The SDK INAP API

For a description of the Intelligent Network Application Part (INAP) protocol standard, see the following website:

http://www.telecomspace.com/ss7-in.html

Within the SDK API, the interaction with ACS for a single call is abstracted into the `ACSSleeTransaction` interface class, which you implement to provide any behavior specific to your application. The control agent creates an instance of this class when it wants to trigger ACS. The Dispatcher then invokes the `handleEvent()` function of `ACSSleeTransaction` when it receives an operation from slee_acs.

The `ACSSleeTransaction` class handles the dialogue primitives `TC-BEGIN` and `TC-CONTINUE` transparently. It also passes each `TC-INVOKE`, `TC-RESULT-L` and `TC-U-ERROR` component primitive to the corresponding receive function.

Your implementation of `ACSSleeTransaction` needs to define the following functions:

- `acs::ACSSleeTransaction::receiveOperation()`
- `acs::ACSSleeTransaction::receiveResult()`
- `acs::ACSSleeTransaction::receiveError()`

Similarly, it also can call these `ACSSleeTransaction` functions:

- `acs::ACSSleeTransaction::queueOperation()`

Passes a `TC-INVOKE` primitive to the TC component sub-layer

- `acs::ACSSleeTransaction::send(bool last)`

  Sends all outstanding components as a `TC-CONTINUE` or `TC-END` operation.

When you call the `send()` function, the `AcsSleeTransaction` object sends a TCAP message containing any queued operations to slee_acs.

The transaction can finish the dialogue by calling `slee::Transaction::end()` or by sending the operations with the `last` flag set to `true`.

The `slee::Transaction::dialogClosed()` function, which the derived transaction class must implement, notifies the transaction when the dialogue finishes.

6

Creating Provisioning Interface Commands

This chapter describes how to create and install new Provisioning Interface commands using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Provisioning Interface Commands

The Provisioning Interface (PI) runs on the Service Management System (SMS) and runs provisioning commands that it receives from a remote source, such as a SOAP request, for example.

The available PI commands and their arguments are defined in the database. You can add new commands using the PI Command Installer utility. For more information, see "Adding a PI Command to the Database".

Each package of PI commands is implemented as a shared library, with one function for each command. Each plugin can contain multiple commands. Using the SDK API, you can create a new plugin that is a package of related commands that the PIprocess process loads on the SMS. Each command in the package is implemented by a function in the shared library.

The PI Function

You must implement each PI command with a function that has the same name as the command. The function's signature must be as shown in the following example:

```
int command_name(char *output, PIinfo *piInfo)
```

where:

- *command_name* the name of the function and matches the name of the command
- *output* is a pointer to a string that stores the output from the command or, alternatively, the name of a file that stores the commands output
- *piInfo* contains the request information being passed to the PI command function

PI Command Actions

Each PI command, and function, can have multiple actions associated with it. For example:

```
COMMAND=Action1
COMMAND=Action2
```

The command function determines which action to execute with code like this:

```
int COMMAND(char *output, PIinfo *piInfo) {

    ...

    if (PIcommonInfo_getAction(piInfo) == "Action1") {
        // Handle Action1
    } else if (PIcommonInfo_getAction(piInfo) == "Action2") {
        // Handle Action2
    }

    ...
}
```

The PIinfo structure is defined in the **PIcommonInfoSDK.h** file and can be accessed by the functions that are defined there.You can find the **PIcommonInfoSDK.h** file in the **$NCC_SDK_HOME/include** directory.

The **PIcommonInfoSDK.h** file is also documented in the SDK API documentation in the **$NCC_SDK_HOME/doc/html** directory.

PI Function Return

The command function should return 0 (zero) if it executes successfully, or an integer code, if it results in an error. You can use the `output` argument to store the commands output. Alternatively, you can store the name of a file that contains the command's output.

You can use the `PIcCommands_success()` and `PIcCommands_error()` helper functions to set up the response.

The following example illustrates a successful outcome:

```
int COMMAND(char *output, PIinfo *piInfo) {

    std::string cmd_name = "COMMAND=" + PIcommonInfo_getAction(piInfo);

    // Processing is successful
    // ...

    std::string responseText = ":Success";

    return PIcCommands_success(output, cmd_name.c_str(), responseText.c_str());
}
```

This example illustrates an error outcome:

```
int COMMAND(char *output, PIinfo *piInfo) {

    std::string cmd_name = "COMMAND=" + PIcommonInfo_getAction(piInfo);

    sms::pi::common::Errors piError;
    sms::pi::common::Errors::codes code;

    // Processing sets an error code
    // ...

    std::string errorText = piError.getText(code);
    return PIcCommands_error(output, cmd_name.c_str(), static_cast<int>(code),
                             errorText.c_str());
}
```

Adding a PI Command to the Database

Follow these steps to define one or more PI commands and add them to the database:

1. Create an XML file that describes the PI commands.
2. Run the **PICommandInstaller** utility, found in the **$NCC_SDK_HOME/bin** directory, to add the file to the database on the SMS.

   The utility will prompt you for the database username and password.

Creating a PI Commands File

You define a PI Commands file by creating an XML file using the XML elements shown in Table 6–1:

Table 6–1 PI Command Definition Elements

XML Element	Description
<PICommandDefinition>	The top level element that contains all other elements.
<Commands>	Contains a <Command> element for each command that you want to define.
<Command>	Contains the elements that define a specific command and action. For example, if a command has three actions like Add, Change, and Delete, then the XML has three corresponding <Command> elements.
<Name>	A string that specifies the PI command (and action) name.
<Package>	A string that specifies a package for the PI command.
<Security>	Optional. An integer that specifies the security level for the PI command. Default value is 1.
<Type>	Optional. A string that specifies the command type. Default value is C.
<Parameters>	Contains a <Parameter> element for each parameter that you want to define.
<Parameter>	Contains the elements that define a specific parameter.
<Name>	A string that defines the parameter name.
<Required>	Specifies whether the parameter is required or not. Possible values are True and False.
<Domain>	A string that specifies the domain for the parameter.

Example: PI Command Definition File

The following statements define several PI command examples:

```
<PICommandDefinitions>
 <Commands>
  <Command>
    <Name>MY_CMD1=ADD</name>
    <Package>SDK_MY_CMD1</Package>
    <Security>1</Security>
    <Type>C</Type>
    <Parameters>
      <Parameter>
         <Name>PRODUCT</Name> <Required>True</Require> <Domain>S</Domain>
      </Parameter>
      <Parameter>
```

```
        <Name>STATUS</Name> <Required>False</Require> <Domain>S</Domain>
      </Parameter>
    </Parameters>
  </Command>
  <Command>
    <Name>MY_CMD1=CHG</name>
    <Package>SDK_MY_CMD1</Package>
    <Security>1</Security>
    <Type>C</Type>
    <Parameters>
      </Parameter>
         <Name>PRODUCT</Name> <Required>True</Require> <Domain>S</Domain>
      </Parameter>
      <Parameter>
         <Name>STATUS</Name> <Required>True</Require> <Domain>S</Domain>
      </Parameter>
    </Parameters>
  </Command>
  <Command>
    <Name>MY_CMD1=DEL</name>
    <Package>SDK_MY_CMD1</Package>
    <Parameters>
      <Parameter>
         <Name>PRODUCT</Name> <Required>True</Require> <Domain>S</Domain>
      </Parameter>
    </Parameters>
  </Command>
 </Commands>
</PICommandDefinitions>
```

Running the PICommandInstaller Utility

You load PI Command definitions into the database by running the **PICommandInstaller** command-line utility, which is in the **$NCC_SDK_HOME/bin** directory. The utility reads a set of PI command definition parameters from the XML file that you created and loads the definitions into the SMS database.

The utility can create and delete PI command definitions, and update the definitions of existing commands.

The **PICommandInstaller** utility has the following syntax:

```
PICommandInstaller -I=<InstallMode(true/false)> [options] <XML file>
```

Where:

- **InstallMode** is either true to install to install the commands in the input file or false to uninstall the commands.
- **options** are:
  - **-F** Optional. Upgrade flag; true to upgrade commands or false otherwise. Default is false.
  - **-v** verbose mode
  - **-h** or **?** for command line help
- *XML file* is the name of the PI command definition file

7

Creating Provisioning Screens

This chapter describes how to create provisioning screens using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Creating Provisioning Screens

You can create new screens with minimal use of NCC classes You can use the `UserScreens.Session` class to create an instance of `java.sql.Connection`. Having created this object, you can create new screens using the Java JDBC and Swing libraries, independent of any NCC code.

Creating Screens Using KFramework

Create new Java screens by following the general look and feel that is used by existing services that are based on KFramework in the core SMS system. You can access the compiled KFramework classes here:

$NCC_SDK_HOME/jar/sms.jar

The classes belong to the UserScreens.KFramework package, of which the main classes are:

- `DataEntryFrame`: The main window
- `DataEntryPanel`: A tabbed panel placed into DataEntryFrame. For example, the sample WindowA, which you can see in Figure 7–1, has two tabs, each of which is a DataEntryPanel.
- `FindDisplayPanel`: A panel that displays the results of find operations

These three classes provide most of the required capabilities and you can extend them through inheritance as necessary.

Follow these steps to make the new service available in the SMS, Services menu.

1. Provide a class named *`service_name.service_name`* that implements the SMS `UserScreens.ServiceScreen` interface.

   The interface contains the following two methods:

   ```
   public abstract MenuItem getMenuItem();
   public abstract void KillAll();
   ```

   The getMenuItem() method creates the menu item on the **Services** menu. The class must implement an ActionListener for the menu item to activate the screen when the user clicks on the menu item.

Use the killAll() method to close down the screens.

2. Configure the following SMS database tables related to screen permissions:
   - SMF_APPLICATION_PART
   - SMF_APPLICATION_ACCESS
   - SMF_TEMPLATE
   - SMF_TEMPLATE_ACCESS

   Note: the database installer, **cmnTableInstaller.sh** configures the SMF_APPLICATION and SMF_APPLICATION_TABLE tables.

   For more information on configuring screen permissions, see "Creating a New Service Screen".

Using the Service Screens

The following sections describe the existing service screens. The base classes provide much of the capabilities described. You can override a significant portion, however, if necessary.

The SMS system integrates the service-specific user screens and they use the SMS security model to provide access to the various onscreen components.

Each database table that a service uses is normally associated with a data entry screen that allows you to create, view, search, modify, and delete entries in the table.

Each data entry screen has the following three modes of operation:

- Find, in which the user can enter query criteria. For more information, see "Find Mode".
- Display, in which the screen displays the results of a query. For more information, see "Display Mode".
- Data Entry, in which the user can create a new record or modify an existing one. For more information, see "Data Entry Mode".

Note: Oracle recommends that you design screens to be compatible with a screen resolution of 1024 x 768.

Find Mode

The find mode is the initial mode for a screen. The screen layout is similar to data entry mode and allows the user to specify criteria to query the database. For components where the user can select from a limited range of values, the user can select an Any or a Don't Care option to specify the values to be matched as follows:

- Combo boxes and List components have an additional selection for the Any value.
- Checkboxes are implemented as tri-state components where the third state indicates the Any value. The three checkbox states are:
  1. White background and unchecked equals not selected
  2. White background and checked equals selected

3. Grey background and checked equals Any or Don't Care value, which is different than disabled and greyed out.

In this mode, each data entry panel contains the following two fields, located at the bottom of the panel, which the user can use to query:

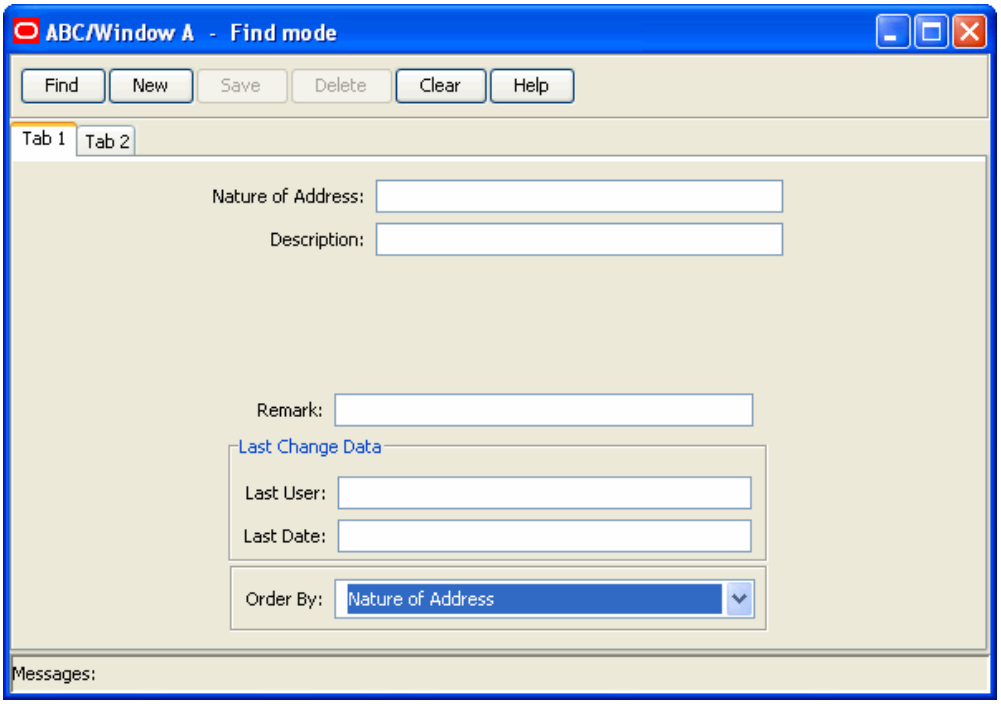
- Last Change User: The user who last modified the selected record
- Last Change Date: The date that the record was last changed

There is also an Order By drop-down box that contains the names of fields that the user can use to specify the criteria for the query. The value that is currently selected determines the initial sorting order for the results. The first entry in the list is the default value, which is always the primary key.

Note: The Last Change Data fields - Last User, Last Date, and Order By - are automatically generated by the base class.

Figure 7–1 illustrates Window A, Tab 1, in find mode.

Figure 7–1 ABC Window A Find Mode



Display Mode

The result panel displays the results of a query when it returns more than one record. It consists of the following two components:

- Results display table
- Find button bar

The result panel is automatically generated and is associated with a specific data entry panel. The results panel maintains a local variable that holds a Java JDBC `ResultSet` object. This variable is defined as an input parameter when the panel is created or

updated by its associated data entry panel. It is scrollable to facilitate the implementation of the **Next** and **Previous** buttons on the **Find** button bar.

The column widths of the result display table will be one of the following standard values:

- Narrow column, 50 pixels
- Small Column, 150 pixels
- Wide Column, 250 pixels
- Widest Column, 400 pixels

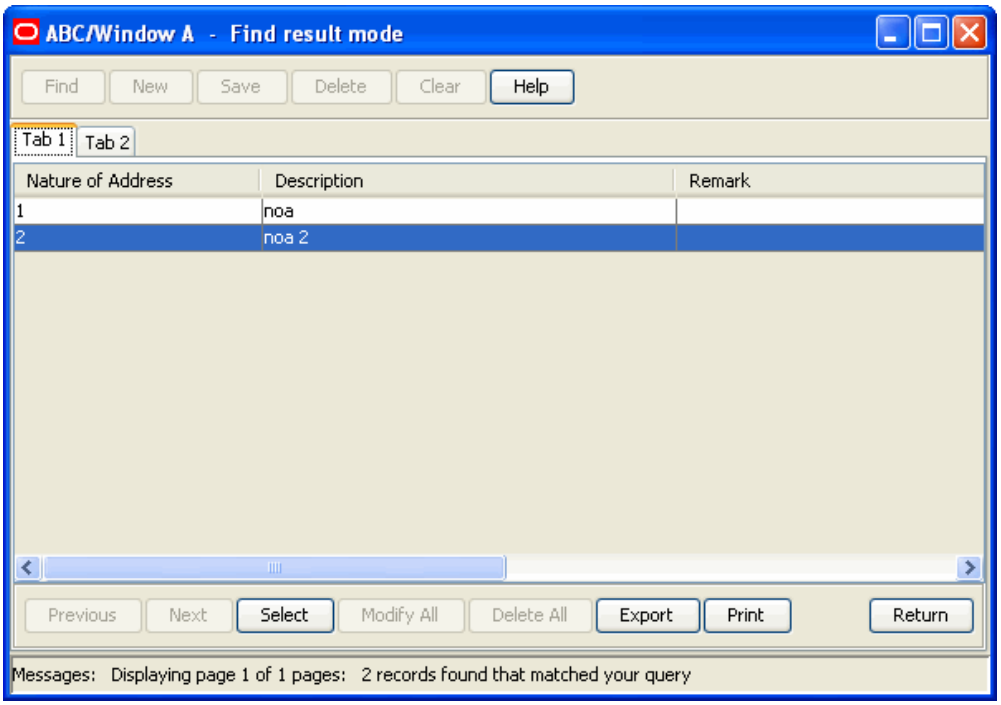
The custom panel determines which value is used by calling the findDisplayPanel.setHeadersAndColumnWidths() function from its initGUI() function. The following example shows the syntax of findDisplayPanel.setHeadersAndColumnWidths():

```
findDisplayPanel.setHeadersAndColumnWidths(namesArray, columnWidths, valuesArray);
```

The results display panel *does not* update automatically to display changes to the database that other users make. The results panel automatically refreshes, using the existing query criteria, after returning from the following operations: modifying a record, deleting a record, a modify-all action, or a delete-all action.

Figure 7–2 illustrates Window A, Tab 1, in display mode.

Figure 7–2 ABC Window A Display Mode



The Results Display Table

The results display table is a sub-class of the Java Swing JTable component and is configured to display a maximum of 100 rows. If the query returns more than 100 rows, the table displays the first 100 rows and also displays a message that indicates the current page position and the total records found, for example, "Displaying records 301-400 of 1234."

If the number of rows exceeds the available space to display them, a vertical scroll bar displays to allow you to display the rows that are hidden.

If the width of the table is greater than the viewing area, a horizontal scroll bar appears to allow you to display the columns that are hidden. Each column in the table has the same width as its related component in the Data Entry panel and the column's name is the same as the component's label.

The component extends the capabilities of the `JTable` class to add the ability to sort the result records by clicking the column header. Clicking a second time reverses the sorting order. Sorting only applies to the currently displayed data and does not affect any data that could be obtained through the **Next** and **Previous** buttons.

You can change the order of a column by clicking on its header and dragging the column to its new position. The ability to auto-resize columns is disabled.

The Find Button Bar

The results display panel has a button bar that displays whenever the panel is visible. The bar is a fixed height and the width of the display area. It contains the buttons listed in Table 7–1:

Table 7–1 Button Bar Buttons

Button	Description
Previous	Enabled if the result set contains more than 100 rows. Displays the previous page of records. Disabled on the first page of the result set.
Next	Enabled if the result set contains more than 100 rows. Displays the next page of records. Disabled on the last page of the result set.
Select	Transitions to data entry mode and replaces the results panel with the data entry panel, which it populates with the data from the currently selected row in the result set. **Note:** Double-clicking a row in the results table has the same effect.
Modify All	Updates records returned by the current query. Displays a dialog box from which you can select fields that you want to modify. Takes you to data entry panel to enter new values for selected fields. Pressing the **Save** button displays a confirmation message in a popup window, indicating how many records will be modified. The window has **Modify-All** and **Cancel** (default) buttons. If any record cannot be modified, any modified record is rolled back and all records are unchanged.
Delete All	Deletes all records returned by the query. Displays a confirmation message in a popup window, indicating how many records will be deleted. The window has **Delete-All** and **Cancel** (default) buttons.
Export	Writes the records found by the query to a file. Displays a standard Java **JFileChooser** dialog box from which you can select a file location. The location defaults to **C:\TEMP.** Record fields are exported in comma-separated format with records separated by Carriage Return / Line Feed (CR/LF).
Print	Prints the records found by the query.
Return	Closes the results display panel and releases any resources used by the screen. Reverts to find mode and leaves the search criteria unchanged.

You cannot add or delete buttons from the button bar.

The Modify All Selection Dialog Box

The Modify-All Selection dialog box displays when the user presses the **Modify-All** button in the Find button bar. In the Available list, the dialog box initially displays the names of each data entry field that you can change with the **Modify-All** command. The Selected list is initially empty.

You can move entries between the Available and the Selected lists, which are mutually exclusive, using the following four buttons:

- > to move the selected item from the Available list to the Selected list
- >> to move all items from the Available list to the Selected list
- < to move the selected item from the Selected list to the Available list
- << to move all items from the Selected list to the Available list

Pressing **Cancel** aborts the Modify-All command and closes the dialog box, leaving the database unchanged. The results panel remains visible and displays the same data that it contained prior to pressing the **Modify-All** button.

Pressing **OK** updates any changes you have made and closes the dialog box.

Data Entry Mode

Use data entry mode to create a new record or to modify an existing one. The screen layout is the same as it is in find mode, except for the following differences:

- The Order By field is not present
- The Last Change User and the Last Change Date fields are greyed out
- Combo boxes, list components, check-boxes, and radio buttons do not have an Any or Don't Care option

Help Screen

The query screens use the existing help mechanism that is part of the of the SMS system. This mechanism uses the Java Help System and incorporates simple help pages in Hyper Text Markup Language (HTML).

Each service screen has its own help screen, which is a single, simply formatted page of HTML text that describes how that screen behaves.

Table Monitor

You can use a simple notification system to enable a data entry screen to notify other screens when a change has been made. Using the table monitor, a data entry screen simply registers its interest in being informed when specific database tables are modified. When a user uses a data entry screen to modify data in the associated database table, the screen simply passes a `tableChanged` message to the table monitor. The table monitor then informs any data entry screens that have registered interest in that table that it has changed.

For example, assume that a screen has a dropdown box that is populated with data from database table Y and registers its interest in the table using the table monitor. If a user subsequently uses screen Y to modify table Y, the table monitor simply informs the original screen that the table has changed. Screen X can then update the contents of the dropdown box to ensure that the contents match the data in database table Y.

Note: The system only handles changes made by a user in that session. If users use separate instances of the SMS screens, the table monitor is not able to inform one user of changes made by the other. The table monitor is also not aware of changes made to the database through SQL scripts such as SQLPlus.

Using TableMonitor

There is only one `TableMonitor` object for each SMS session. This object maintains a hashtable where the key is a database table name and the value is a vector of data entry screens that have registered interest in that table.

The TableMonitor class does not have a public constructor. It has the following public interface:

- public static TableMonitor getTableMonitor();
- public void addTableMonitorListener(TableMonitorListener listener, String tableName);
- public void removeTableMonitorListener(TableMonitorListener listener, String tableName);
- public void tableChanged(String tableName);

Use the getTableMonitor() method to obtain an instance of TableMonitor. This method always returns the same TableMonitor instance.

Use the addTableMonitorListener() method to register interest in a database table. Use the removeTableMonitorListener() method to remove an entry for a specific database table. When TableMonitor receives a message from one of these methods, it updates its internal hash table accordingly.

Use the tableChanged() method to inform the TableMonitor that the table specified by the input parameter has changed. When TableMonitor receives a message from tableChanged(), it obtains the vector of TableMonitorListener objects that have registered interest in that table and relays the message to all of the objects within that vector.

A data entry screen must implement the TableMonitorListener interface to be informed by TableMonitor when a database table changes. The `TableMonitorListener` interface has the following definition:

```
public interface TableMonitorListener() {
        public void tableChanged(String tableName);
}
```

Creating a New Service Screen

This section describes the process to create new Java data entry screens for a new service using KFramework. It uses an example in the SDK called ABC to describe the process. The example adds a new **ABC** menu to the SMS Service menu. The menu contains two sub-menus, **Window A**, which has two tabbed panels, and **Window B**, which has one tabbed panel. The service adds three new database tables.

Note: Attempting to use the **abc.jar** file without signing it results in a warning message when you log in to the SMS screens. The procedures that follow in this section describe how to sign the jar file.

To create a new service screen:

1. Design the database schema for the new service.
2. Determine the application name and application ID for the new service.

   The application name should not contain spaces and should be a valid Java variable name. This example uses the name `ABC`.

   The value for application ID should be between 900 and 999, which is the range allocated to custom applications.
3. Configure the screen permissions for the new service by running the following commands:

   ```
   sqlplus user/password
   @$NCC_SDK_HOME/example/ABC/db/addPermissions
   ```

4. Create a new Java class called ABC.ABC.

   This class must implement the UserScreens.ServiceScreen interface that adds the new service to the core SMS **Services** menu. The name must match the value of the APPLICATION column in the SMF_APPLICATION database table, which is ABC in this example.
5. Create a new screen for the service.

   This class is inherited from the UserScreens.KFramework.DataEntryFrame class. Each instance of DataEntryFrame can contain one or more panels (DataEntryPanel instances), which are managed in a JTabbedPane component. The parent class does the majority of the work and usually requires some additional code. It primarily creates instances of panels to be added to the JTabbedPane component.
6. Design and create the panels that need to be added to each instance of `DataEntryFrame`.

   Each panel should extend the UserScreens.KFramework.DataEntryPanel class. Data aware components that read and write to the database should belong to the UserScreens.KFramework.DataEntryPanel class. The base class does a large portion of the work required by the panel but you can override many of the `DataEntryPanel` methods to provide customized behavior. For more information, see "Creating DataEntryPanels Classes".
7. Build the custom *app_name***.jar** file by running the following commands:

   ```
   cd $NCC_SDK_HOME/app_name/java
   gmake install
   ```

8. Create a SHA-256 signed version of the JAR file by running the following commands:

   ```
   cd $NCC_SDK_HOME/bin
   keytool -genkeypair -keyalg RSA -keysize 2048 -alias SDK
   jarsigner -signedjar app_name.jar.sig app_name.jar SDK
   ```

9. Copy the signed JAR file from **$NCC_SDK_HOME/bin** to **/IN/html** on the SMS machine.
10. Create a **/IN/html/sdk.jnlp** file and add the following text to it to describe the custom screens:

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

    <jnlp spec="1.0+" codebase="http://xx.xxx.xxx.xxx/" href="sdk.jnlp">
    ```

```
   <information>
      <title>ABC</title>
      <vendor>Custom</vendor>
   </information>

   <resources>
     <j2se version="1.8.0+" href="http://java.sun.com/products/autodl/j2se"/>
     <jar href="app_name.jar.sig"/>
   </resources>
   <component-desc/>
</jnlp>
```

11. Save and close the file.
12. Open the **/IN/html/sms.jnlp** file and add a reference to the new **sdk.jnlp** file:

    ```
    <extension name="ABC Service" href="sdk.jnlp"/>
    ```

13. Save and close the file.
14. Register the service with SMS by logging into the SMS database as the SMF user and running the following command:

    ```
    $NCC_SDK_HOME/example/ABC/db/addPermissions.sql
    ```

15. Create Oracle error translation and language translation files as needed. For more information, see "Language Translation".
16. Create and install a deployment rule for the SHA-256 signed version of the JAR file that you created by doing the following:
    a. Obtain the SHA-256 hash from the *app_name*.**jar.sig** file:

       ```
       keytool -printcert -jarfile app_name.jar.sig
       ```

       The hash is the 32-pair hexadecimal digit in the line that starts with SHA256.

    b. Create a rule set file with the name **ruleset.xml**. See *Oracle Java Platform, Standard Edition Deployment Guide* for instructions.

    c. Open **ruleset.xml** and replace the certificate hash with the hash that you obtained in Step 14. a:

       ```
       <ruleset version="1.0+">
         <rule>
           <id location="http://xx.xxx.xxx.xxx/">
             <certificate algorithm="SHA-256"
       hash="00:01:02:03:04:05:06:07:08:09:0A:0B:0C:0D:0E:0F:10:11:12:13:14:15:16:
       17:18:19:1A:1B:1C:1D:1E:1F" />
           </id>
           <action permission="run" version="1.0+" />
         </rule>
       </ruleset>
       ```

    d. Create a deployment JAR file with the name **DeploymentRuleSet.jar** and include the **ruleset.xml** file in the **DeploymentRuleSet.jar** file:

       ```
       jar cf DeploymentRuleSet.jar ruleset.xml
       ```

    e. Sign the **DeploymentRuleSet.jar** file with a valid certificate from a trusted certificate authority.

f. On each client machine, install the signed **DeploymentRuleSet.jar** file in the following location:

Linux: /etc/.java/deployment/DeploymentRuleSet.jar

Microsoft Windows:
C:\Windows\Sun\Java\Deployment\DeploymentRuleSet.jar

OS X: /Library/Application
Support/Oracle/Java/Deployment/DeploymentRuleSet.jar

Solaris: /etc/.java/deployment/DeploymentRuleSet.jar

17. Verify that the deployment rule set is installed and signed correctly:

a. On each client machine, open the Java control panel.

b. On the Security tab, the appearance of the "View the active Deployment Rule Set" link indicates that the deployment rule is installed correctly.

To verify the validity of the signing certificate, click the "View the active Deployment Rule Set" link.

The ABC Example

You can find the source files for the ABC screens example at **$NCC_SDK_HOME/example/ABC/java**. The **ABC.java** file defines the ABC package, which contains the ABC class. The ABC class implements the ServiceScreen interface.

The package name and class name must match the name in the APPLICATION column of the SMF_APPLICATION database table because the core SMS system obtains the value when the main applet starts and then uses the Java Reflection API to create an instance with that package name and class name.

Once the ABC object has been created, the example calls the public MenuItem getMenuItem() method to obtain a reference to the new menu object that provides access to the service. If the current user does not have sufficient access permission, the method returns null and the service is not available.

When the main applet closes, it calls the killAll() method, which releases any resources that the service opened.

This example creates a new menu named **ABC,** with two sub-menus, **Window A** and **Window B**. Each window can contain one or two panels. The getMenuItem() method creates a sub-menu only if the user has permission to access at least one of the panels.

Each sub-menu is associated with an ActionListener object, which calls a static start() method on a class that implements the UserScreens.DataEntryFrame class. The start() method creates an instance of the class and makes it visible to the user.

As mentioned previously, you can create new service screens without using **KFramework**. Once you create the database connection using UserScreens.Session.database.connection, you can create new service screens using any of the standard Java, JDBC, AWT and Swing libraries.

Use the UserScreens.General.trace() method to send output messages to the Java console. These messages are visible in the console only if TRACE has been turned on using the **SMS Operator Functions / User Management** screen to edit the user and set the Configuration field to **TRACE=ON**. This method provides a simple but useful debugging facility.

The UserScreens.Language.getTranslatedString() method looks for a translated version for the message in the input parameter. If found, it returns the translated value; otherwise, it returns the original input message.

Creating DataEntryFrame Classes

Each frame for the service extends the UserScreens.KFramework.DataEntryFrame class. This is a relatively simple subclass to implement because the parent class does most of the work. For the ABC example, you can find these classes in the com.oracle.abc.screens package.

Please refer to the **$NCC_SDK_HOME/example/ABC/java** files for the actual source code.

There are two subclasses, AScreen and BScreen. Because each of these is virtually identical, only AScreen will be described here.

The class needs to override the following two methods in the parent class:

```
public static void start();
public static void stop();
```

The ABC.ABC class calls these methods. It calls start() when the user selects the menu item for this service. It creates the actual frame and its associated data entry panels. The ABC class calls stop() in response to the main applet being closed and releases any resources held by the screen.

Most of the remaining code consists of basic Swing library calls that create the required user interface.

The canAddTab() method, which the parent class defines, checks to see if the current user has sufficient permission to access a particular data entry panel.

Creating DataEntryPanels Classes

Each data entry panel for the service must extend the UserScreens.KFramework.DataEntryPanel class. Again, the parent class does most of the work. However, the majority of the work to implement the service screens takes place here.

The DataEntryPanel class contains a large number of methods, many of which you can override, if necessary. For example, the parent class auto-generates the SQL search query, but provides hooks so that you can override parts of the query, such as the WHERE clause, for example, without rewriting the whole query.

The AbcPanelOne example provides a data entry screen for a database table with the columns described in Table 7–2. This table is partly described in the following SDK file:

$NCC_SDK_HOME/example/ABC/db/SDK.xml

Table 7–2 Database Table for AbcPanelOne

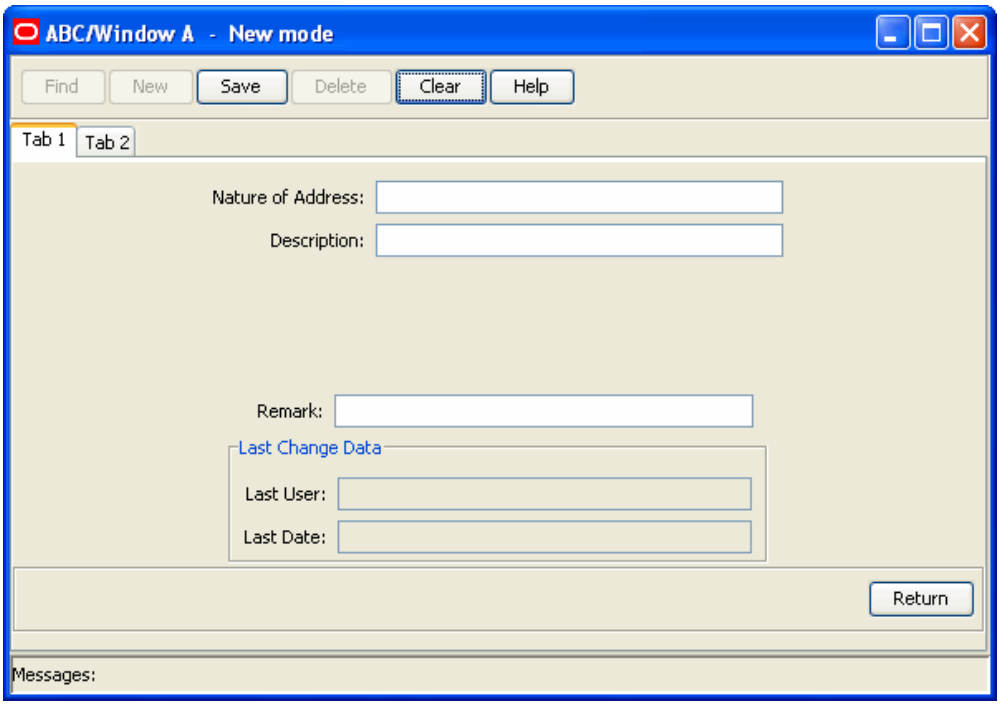
Column Name	Data Type
NOA	NUMBER (3) (Primary Key)
DESCRIPTION	VARCHAR2 (50)
CHANGE_REF	VARCHAR2(50)
CHANGE_TERM	VARCHAR2 (12)
CHANGE_USER	CHAR (50)

Table 7–2 (Cont.) Database Table for AbcPanelOne

Column Name	Data Type
CHANGE_DATE	DATE

Figure 7–3 shows the Window A, Tab1, screen for entering the data described in Table 7–2.

Figure 7–3 Window A Tab1 Data Entry Screen



You can find the source code for this panel at this location:

$NCC_SDK_HOME/example/ABC/java/AbcPanelOne.java

This class is meant to use with components on the panel that are mapped to a single database table.

The Constructor

The example uses the main constructor with four parameters:

```
public AbcPanelOne(DataEntryFrame parent, String newTableName, String appName,
String appPart);
```

The newTableName parameter is the name of the database table to which the screen panel maps. The appName parameter is required to obtain access to the correct error and language translations. The appPart parameter is required for the access permissions.

TableMonitor

The TableMonitor class has been described earlier in this document. This panel registers itself with TableMonitor and then informs it when any changes are made. This requires the save(), delete() and deleteAll() methods of the parent class to be overriden. The example shows how this is done.

Help

The help() method calls the help(String section) method of the parent class. The section parameter provides the key into the Help system.

Validation

The focusGained() and focusLost() methods place a simple message into the message display area whenever the `noaField` has focus. The message displays the range of permissible values.

The addDocuments() method adds `DocumentListeners` to the two main text fields to restrict the input to a range of valid values. In the case of `noaField`, these values are numeric digits in the range of 0 - 127 only. The `descriptionField` field can have any characters up to a maximum length of 50.

The GUI

The initGUI() method creates the user interface:

The parent class displays the **Find Display Panel**. The parent class could obtain the database table metadata automatically, but it needs to know what names to use for the column headings and the column widths. Assigning a display name can sometimes be more descriptive than the column name. The findDisplayPanel.setHeadersAndColumnWidths() method allows you to do this.

The parent class automatically generates the **Change Reference**, **Change User** and **Change Date** components.

Place the non auto-generated components into the `customPanel` variable;

You can use data-aware components for optional or mandatory values. If a field is mandatory, you can call the setNotNullFlag(true) method on that component. In this case, the parent class checks that a value has been provided before saving. If a value has not been provided, a warning dialog displays.

The auto-generated **Order By** component allows the user to choose how to display the results of a query. Use the commonPanel.setOrderByComboBoxValues() method to populate the component. The selected component requires two arrays - one for database column names and the other for the associated display name.

Call the customPanelModified() method after making any changes to **customPanel**.

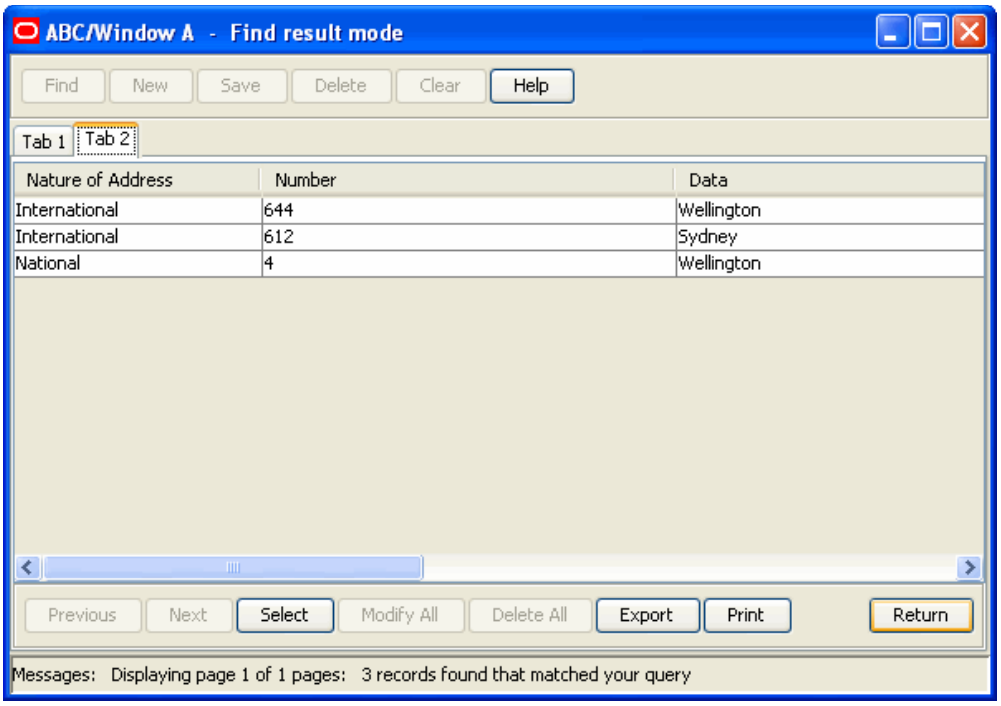
The final line of the initGUI() method ensures that the panel is in the correct starting state

Some database tables are replicated and some are not. This has an impact on the data entry screens. For tables that are replicated, you are not allowed to update their primary keys. In this example, the table is not replicated. If a table is replicated, call the setReplicated(true) method, which causes the parent class to disable the primary key fields when updates occur, and the **Check Consistency** button becomes enabled. This button is only enabled for replicated tables.

There are no foreign key fields in this example. Foreign key fields cause some complications for the print and export mechanisms. Often the foreign key column contains an ID value such as a number. This is not particularly meaningful for the user, so it is possible to map the ID value to a more descriptive display value. You can do this with a simple array or through a database join.

The Tab 2 panel, **AbcPanelTwo.java**, for the same window is a slightly more complicated example. Figure 7–4 illustrates the display mode for the Tab 2 panel of Window A.

Figure 7–4 Window A, Tab2 in Display Mode



You can find the source code for this panel at this location:

$NCC_SDK_HOME/example/ABC/java/AbcPanelTwo.java

Table 7–3 shows the database table for this screen:

Table 7–3 Database Table for AbcPanelTwo

Column	Description
NR_NOA	NUMBER (3) (Primary Key)
NR_NUMBER	VARCHAR2 (20) (Primary Key)
NR_DATE	VARCHAR2 (50)
CHANGE_REF	VARCHAR2 (50)
CHANGE_USER	CHAR (50)
CHANGE_DATE	DATE

While this screen does not look much different from the previous one, there are some differences in the code. The NR_NOA column is a foreign key into the SDK_ABC_NOA database table. Therefore, the component associated with this column provides a mapping between the actual foreign key (NR_NOA) and the SDK_ABC_NOA.DESCRIPTION value.

Note that this time the class implements the TableMonitorListener interface. This allows the panel to update the **Nature of Address** component whenever the panel in **Tab 1** creates, modifies or deletes an entry. The constructors make an additional call to register interest in the SDK_ABC_NOA table with tableMonitor.

```
tableMonitor.addTableMonitorListener(this, NOA_TABLE_NAME);
```

The TableMonitorListener interface has just one method, which is shown here:

```
public void tableChanged(String tableName);
```

This method updates the **Nature of Address** component whenever it is called. The following line of code does the update, forcing the component to refresh itself with the latest database information:

```
noaComboBox.reReadDataBase();
```

Because we have registered an interest in another table, we must de-register when we close the window frame. Do this by overriding the public void close() method.

The createCustomInnerPanel() method is straightforward.

The noaComboBox component is implemented within a `try` and `catch` block as a database query is required to populate the component. This time a simple **GridBagLayout** is used to layout the components.

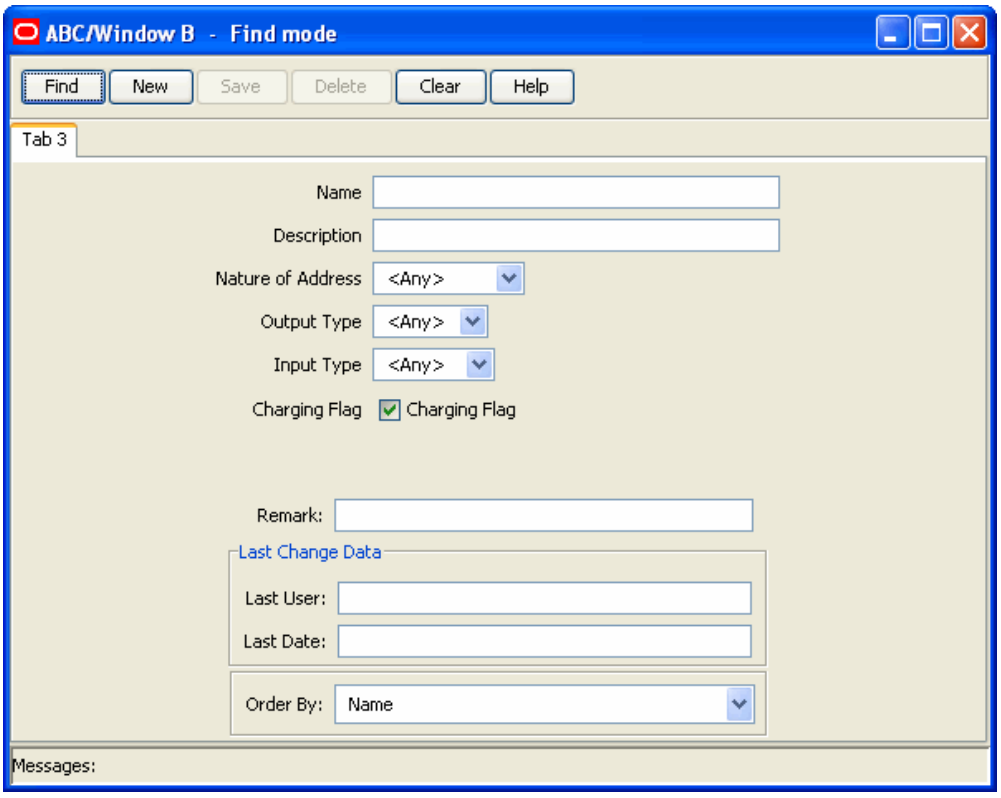
The initGUI() method is virtually the same as in the previous example except that it has the following additional line:

```
setSupplementaryQuery(supplementaryQueryArray);
```

This line is part of a mechanism that allows the screen to map a foreign key value to a more meaningful value, in this case mapping the noa number to its description. This allows the **Find Display Panel** to show the NoA description, which is probably more meaningful to the user, instead of a number.

The following screen shot shows the final tab panel, on the Window B frame, which has some additional differences.

Figure 7–5 Window B in Find Mode



The full source code for this example is available in the following location:

$NCC_SDK_HOME/example/ABC/java/AbcPanelThree.java

This time the **Output Type**, **Input Type** and **Charging Flag** components map to database columns that are of type `CHAR(1)` and have the following values.

- Input Type: Has valid values of I and E which represent Internal and External respectively
- Output Type: Has valid values of A, B and C which represent Output A, Output B and Output C. This employs the technique of using the component to map to an array as opposed to a foreign key table relationship
- Charging Flag: Has valid values of T and F which represent true or false

The source code shows how you can use an instance of **ScreenComponentFilteredComboBox** to create a set of key value pairs that map the database values to the values that are displayed to the user.

You can create instances of **ScreenComponentTriStateCheckBox** to provide a mapping to various database values. This is done using the constructor as shown in the following example:

```
flagCheckBox = new ScreenComponentTriStateCheckBox("CHARGING_FLAG", false, true,
       true, CHARGING_FLAG_STRING, TriStateCheckBox.NOT_SELECTED, "T", "F", null,
        "Active", "Not Active", "Don't Care");
```

In this case, the component is mapped to T when the checkbox is selected, and to F when it is not selected. Note that these values could also be Y or N for yes or no.

The component also has a third state ("Any" or "Don't Care") for when the panel is in find mode. It allows the search facility to match against any value.

Another difference in this screen is that the underlying database table uses an **ID** column as the primary key, which is auto-generated by an Oracle sequence. Normally this value is of little importance to the user so it is not usually displayed. This does have a couple of consequences, the main one being whether the CheckConsistency() method needs to be overridden. In the following code, the **Name** and **NoA** fields are unique so they are used to obtain the primary key value (the `ID`) for this entry. Note that the **CheckConsistency** button is only available when a database entry has been selected for update.

This time, the initGUI() method uses a slightly different setSupplementaryQuery() method call:

```
setSupplementaryQuery(supplementaryQueryArray, supplementaryDataArray,
      supplementaryDisplayArray);
```

The `supplementaryDataArray` and `supplementaryDisplayArray` parameters provide mapping information to the **Find Display Panel** for the components that have hard-coded lists (arrays), for example the **Input** and **Output Type** components.

Other useful methods in the parent class that you can override with subclasses are:

- The search mechanism comprises multiple methods and is used to automatically generate the SQL search query, the main one being shown here:

  ```
  protected String createFindQueryString(ScreenComponent[] screenComponents)
  ```

  The input parameter is an array of components that contain search values. This method makes use of a number of helper methods to generate the different parts of the query, for example, the SELECt clause, FROM clause, WHERE clause and ORDER BY clause.

The helper methods are:

- getSelectionList(): To add columns from a join or add an Oracle hint
- getFromClause(): To include extra join information
- getWhereClause(): To exclude certain values, for example where ID > -1
- getOrderByClause(): To restrict to a subset all the fields that can be used in the ORDER BY clause
- getFindableDateWhereClause (ScreenComponentIndentedFindableDate comp): A findable date uses two parts, for example date >= ? and date <= ?. Normally the find query performs an exact match, for example start_date = ?
- getExportFindQueryString(): By default, returns the same value as a call to createFindQueryString(). You can override it, though, if that is required for exporting data
- getExportFindQueryString() : The default implementation returns the same value as a call to createFindQueryString(), but can be overridden if this is required for exporting data
- protected PreparedStatement getExportStatement (): To return the a prepared statement that performs the same query as a find that returns a scrollable result set but instead uses a non-scrollable result set for use in exporting tables with a large number of entries.

The save mechanism also consists of a number of methods that you can be override as required:

- public boolean save() : This is the main entry point for the save mechanism
- protected boolean canSave(): This method is called from the save() method and provides verification and validation of input data. The method should return `true`, if all of the data entry fields contain valid data. Typical examples are mandatory fields that have a value and data that is within specified limits. This often needs to be overridden when one field is used in conjunction with another field, such as using minimum and maximum pairs to check that the minimum value is less than the maximum value. Another common reason to override this method is when a field is only mandatory when the value of another field is set to a particular value (for example, a check box is selected).
- protected int doInsert(): This method performs the actual INSERT statement required to create a new database record. It returns the number of rows inserted, which will normally be 0 or 1, but it is conceivable that other values could be valid in some circumstances.
- protected int doUpdate (): Similar to the doInsert() method, but this updates existing records in the database rather than create new records.
- protected String getUpdateSetClause(): This method generates the SET clause for the UPDATE statement. By default this includes all onscreen components that implement the ScreenComponent interface, are enabled and can be saved.
- protected int doModifyAllSave(): This method performs the actual UPDATE statement required by a Modify All operation where the same updates are performed on multiple rows.
- public Vector getModifyFields(): This method is used by the Modify All mechanism. It contains a list of the fields that a user may modify. This would typically exclude any primary key fields. Otherwise the result would be a UNIQUE constraint violation.

- protected String getUpdateSetClause(): This method generates the SET clause for the UPDATE statement. Again, by default, it includes all on-screen components that implement the ScreenComponent interface and are enabled and can be saved.
- delete() and deleteAll() : If the subclass needs to use these methods, they can be overridden. This is usually done when the class needs to work with the TableMonitor class.
- public void dispose(): The default implementation is an empty stub, but this can be used to free any held resources, if required.
- public void close(): In the default implementation this closes any open result sets or FIND statements. As the results sets are scrollable, they are not closed immediately. You must ensure that they are closed when the screen is no longer required.
- public boolean canClose(): This method determines whether a screen can close. The default implementation checks to see if the panel data has been modified but not saved. The method returns `true` if it is alright for the panel or window to close, and `false` otherwise.
- public void setStatusMessage (String) and public String getStatusMessage(): These two methods set and retrieve the contents of the message display area.
- public void focusGained(FocusEvent) and public void focusLost(FocusEvent): The default implementation for these two methods does nothing. However, you can override them to specify any behaviour that needs to occur when the panel gains or loses focus on the screen.
- protected void setSupplementaryQuery(): Part of a very complicated mechanism that ensures that the findDisplayPanel shows descriptive key data in the display table. This is also used by the export operation.

Language Translation

The ABC example uses the SMS language translation mechanism. Statements similar to the one shown in the following example appear at various points in the source code:

```
String translatedMsg = Language.getTranslatedString(String application, String
message);
```

In this example, `application` will have a value of `ABC` and the `message` parameter is the text that needs to be translated. The SMS system searches for a translated version of the message text and, if found, returns it. Otherwise, it returns the original message.

The mechanism uses a flat, plain text file containing a list of key value pairs with the format:

```
<key>=<translated value>
```

where `key` is the message in the default language on the left-hand side and the translated version is on the right-hand side. For example:

```
Hello=Bonjour
```

The file must reside in the directory **/IN/html/<Application Name>/language** and the name must have the format **<language name>.lang** as shown in the following examples:

```
/IN/html/ABC/language/English.lang
/IN/html/ABC/language/Dutch.lang
```

In the **English.lang** file the key value pairs would be identical.

For the ABC example, a **strings.txt** file contains a list of all the strings that need to be translated. You can find the **strings.txt** file in **$NCC_SDK_HOME/example/ABC/html/**.

You can use the Unix shell script, **regen-lang** to generate the **English.lang** file with the following command:

```
$ ./regen-lang > English.lang
```

The **regen-lang** script can also generate a useful dummy language file by using the **d_** parameter, as shown in the following example:

```
$./regen-lang d_ > Dutch.lang
```

This form of the command produces a new **Dutch.lang** file in which the right-hand side of each key value pair is the same as the left-hand side, but is prefixed by **d_**. For example:

```
hello=d_hello
```

You can use this option to quickly see if any translations have been missed. Simply set the SMS language configuration to Dutch and every visible string in the service should be prefixed by **d_**. If not, then you need to add that string to the file.

SQL Error Translation Files

The service screens can generate an SQLException, for example, when trying to create a new record that results in a non-unique constraint error. If the SQL error message that is produced by JDBC and the Oracle drivers is not descriptive enough, you can provide a more descriptive message. If a new service introduces new database constraints that users are likely to encounter, then you must provide an error message and translation.

The translation files go in the directory **/IN/html/<ApplicationName>/error/<language name>**. For example, for an application named ABC and the English language, the files would go in **/IN/html/ABC/error/English**.

The name of the file that contains the text for the error code must be either **<error_code>** or **<error_code>.<detail>**. The **<detail>** part allows you to have different messages for the same error code. For example, error code 1, a unique constraint violation, indicates a different problem for each constraint that might be violated.

In this case, the SQL error is parsed and the part in brackets after the first dot is taken as the detail field. For example, given the SQL error "ORA-00001: unique constraint (SMF.SMF_APP_PK) violated", the detail part will be "SMF_APP_PK", so the error file should be named **1.SMF_APP_PK**.

The ABC example contains several examples of message files in the **$NCC_SDK_HOME/example/ABC/html** directory.

Service Help Files

The SMS system and the services that its supports use the Oracle Help for Java system. For details on using the system, please refer to the official Oracle Help for Java documentation.

The help file name should be **<language>_<application>.hs**.

For the ABC example, you must create the following directory on the SMS server.

/IN/html/ABC/helptext

This directory must contain the file **/IN/html/ABC/helptext/English_ABC.hs**

This file tells the Help system where to find the rest of the help files, in particular the **map.jhm** and **toc.xml** files.

The following example shows the **English_ABC.hs** file. For a new service, all that you would need to change is the title and the homeID value.

```
<?xml version='1.0' ?>
<helpset version="1.1">

<!-- title -->
<title>ABC Helpset</title>

<!-- maps -->
<maps>
    <homeID>056789</homeID>
    <mapref location="English/map.jhm" />
</maps>

<!-- views -->
<view>
    <name>TOC</name>
    <label>Contents</label>
    <type>oracle.help.navigator.tocNavigator.TOCNavigator</type>
    <data engine="oracle.help.engine.XMLTOCEngine">English/toc.xml</data>
</view>

<view>
    <name>Index</name>
    <label>Index</label>
    <type>oracle.help.navigator.keywordNavigator.KeywordNavigator</type>
    <data engine="oracle.help.engine.XMLIndexEngine">English/index.xml</data>
</view>

</helpset>
```

For details on how to create the actual help content, including the **map.jhm** and **toc.xml** files, refer to the Java Help documentation.

8

Creating Memory-Mapped Files

This chapter describes how to create memory-mapped files using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Memory-Mapped Files

The NCC SDK enables you to create memory-mapped files, which you can use to increase application performance. Memory mapped files allow you to map data from database tables to files that are memory resident. This provides fast database access for components that reside on Service Logic Controllers (SLCs), such as feature nodes and service loaders. Applications that have the following characteristics can benefit from using memory-mapped files:

- Require total data availability
- Require high rates of random reading
- Have data that rarely changes, such as routing tables and product-related tables

About Creating Memory-Mapped Files

The creation of memory-mapped files consists of the following processes:

- Data must be transferred from the Service Management System (SMS), where the master database resides, to the Service Logic Controller.

  The data is transferred using the SMS replication facility.

- Changed data must be detected

  This includes detecting the absence of change in the data and also the signalling of an alert on each SLC to indicate when the data has changed.

- Providing the data to the application

The following steps describe the process in more detail:

1. Table X in the Oracle database is updated on the SMS.
2. The SMS replicates the table to the SLC.
3. When the updates are complete, an insert is done to the **SMF_APPLICATION_ALERT** table on the SMS.
4. SMS replicates the change to the **SMF_APPLICATION_ALERT** table on the SLC
5. An Oracle alert triggers an Mfile daemon

6. The Mfile daemon reads the entries from Table X, formats them, and writes them to a file.
7. The file is mapped and read by the application.

Data Replication

SMS replicates data changes to the SLC just as it normally does. SMS replicates changes to the database and updates to the **SMF_APPLICATION_ALERT** table automatically. No development is required to accomplish the replication of data changes and alerts to the SLCs. The **cmnTableInstaller.sh** utility allows you to configure the replication and create the replication triggers.

For information on using the **cmnTableInstaller.sh** utility, see "Running the Database Table Installer" in Chapter 9.

Creating Alerts When Data Changes Occur

On the SMS, inserts to the **SMF_APPLICATION_ALERT** table will be deleted automatically and the insert and delete are replicated to all SLCs. Only inserts are audited; updates are not permitted.

On the SLC, each insert causes an alert to be signalled.

For an application named **myApp** and a table named SDK_ANUMBER_BARRED the following SQL code must be executed on the SMS to ensure any change within the data produces an insert-and-delete cycle on the SMF_APPLICATION_ALERT table, which signals the associated Mfile daemon to detect the change and rebuild the Mfile.

```
create or replace trigger smf.SDK_ANUMBER_BARRED_mf_auid
after insert or update or delete on smf.SDK_ANUMBER_BARRED
referencing new as r for each row
begin
insert into smf_application_alert (application, item) values ('SDK', 'SDK_ANUMBER_
BARRED');
delete from smf_application_alert where APPLICATION='SDK' and ITEM='SDK_ANUMBER_
BARRED';
end;
```

An Mfile daemon can register for these alerts and respond by regenerating the data or following other instructions.

The Mfile Daemon

The daemon that creates the memory-mapped file runs on each SLC. It takes the following parameters:

- mapped file name
- application and item
- date query
- data regeneration function

The SDK provides classes and functions that allow you to create the Mfile daemon and organize the Mfile data in a number tree fashion. You can have multiple trees with each tree referenced by an index. You can elect to store three integers or a block of data at each leaf on the tree. Typically, each block of data is derived from a row in a specific database table. Leaves in containing data blocks are referenced by a unique string.

The SDK provides functions to build the tree and to add to it, as well as to search the tree. The type of storage and searching that is used is referred to as General Purpose Number Analysis (GPNA). Each block of data typically is derived from a row in an application-specific database.

The Mfile Daemon API

The SDK provides functions that allow you to create daemons to retrieve information from database tables and create an Mfile.

The **cmnMfileSDK.h** and **cmnMfileDaemonAPI.h** files define the prototypes for these functions. The static library **libcmnMfile.a** implements them.

The **cmnMfileDaemonAPI.h** file includes the functions described in the following sections.

enum AwaitResult{...}

Several of the functions in `cmnMfileDaemonAPI.h` return `enum` AwaitResult{...}, which is defined as follows:

enum AwaitResult(...)

where the possible values are:

- `AWAIT_FAILED` = -1 if a catastrophic error occurred
- `AWAIT_CHANGED` if the data has changed, or the file is invalid
- `AWAIT_EXIT` if the daemon was asked to exit

initGPNA()

The initGPNA() function initializes the specified Mfile for the specified database file. You call it once at the start of an Mfile daemon.

This function has the following syntax parameters, and return value:

```
int initGPNA(char *userpass, char *dbtable, char *app_name, char *filename,
          int init_sz)
```

Parameters

- `userpass` is the Oracle username and password
- `dbtable` is the name of the database table to map to the Mfile
- `app_name` is the application name
- `filename` is the Mfile file name, including the full path
- `init_sz` is the initial size to make the Mfile

Return

- 0 if successful
- -1 if creation of the Mfile fails

awaitGPNAChange()

The awaitGPNAChange() function checks to see if a file is up to date and waits for an appropriate alert or exit request through the Mfile API. This function blocks until there is a change on the database table that was initialized by initGPNA().

This function has the following syntax and return values:

```
enum AwaitResult awaitGPNAChange()
```

Return

- 0 if successful
- -1 if failure

startGPNAChange()

The startGPNAChange() function forces reload of the Mfile but does not depend on a change in the database. It is a non-blocking function.

This function has the following syntax and return value:

```
enum AwaitResult startGPNAChange()
```

Return

- 0 if successful
- -1 if failure

mallocGPNAEntry()

The mallocGPNAEntry() function retrieves the next available memory position at which to write the next GPNA entry, returning a pointer to this location. You can then write a block of data there, not to exceed the limit specified by the `max_entry_size` parameter.

This function has the following syntax, parameters, and return value:

```
void *mallocGPNAEntry(int max_entry_size)
```

Parameters

- `max_entry_size` is the maximum size of the memory block to allocate

Return

- pointer to available memory block for leaf data, if successful
- NULL, if failure

addGPNAEntry()

The addGPNAEntry() function adds a leaf entry at the location returned by the call to the mallocGPNAEntry() function.

This function has the following syntax, and parameters.

```
void addGPNAEntry(int size_of_entry, unsigned char *nature, char *number)
```

Parameters

- `size_of_entry` is the actual size of the entry, which must be the same as allocated through mallocGPNAentry()
- `nature` is a pointer to the Nature of Address value, an integer that describes the type of number, for example, national or international.
- `number` is a pointer to a string that contains the digits of the address

addGPNAIntEntry()

Adds a `GPNAResult_t` value element to the uncompressed tree.

This function has the following syntax and parameters:

```
void addGPNAIntEntry(GPNAResult_t res, unsigned char *nature, char *number)
```

Parameters

- `res` is the `GPNAResult_t` value element to add to the uncompressed tree. `GPNAResult_t` is the result of the genericGPNA() function.
- `nature` is a pointer to the Nature of Address value, an integer that describes the type of number, for example, national or international
- `number` is a pointer to a string that contains the digits of the address

finishedGPNA()

Call the finishedGPNA() function after all entries have been added to the uncompressed tree. This function compresses the GPNA tree, frees up memory, and replaces the shared memory version of the table with the version just created.

This function has the following syntax and return value:

```
int finishedGPNA(void)
```

Return

- 0 on success
- -1 on failure

finishedSingleEntry()

Call the finishedSingleEntry() function after adding a single entry to the uncompressed tree. This function compresses the tree, and frees up memory.

This function has the following syntax and return value:

```
int finishedSingleEntry(void);
```

Return

- 0 on success
- -1 on failure

You can find an Mfile daemon example in the following SDK file:

$NCC_SDK_HOME/example/sdkMfileDaemon/sdkMfileDaemon.cc

An Mfile Daemon Example

You can find an Mfile daemon example in the following SDK file:

$NCC_SDK_HOME/example/sdkMfileDaemon/sdkMfileDaemon.cc

The Mfile Application

The SDK also provides functions to allow you to create your own APIs to retrieve information from memory-mapped files. You can use these APIs in service components to retrieve information as rapidly as possible.

Before you access an Mfile from your application, you must create a global mapping of the memory table identifiers to the shared memory files. You must include code in your application that calls the setupGPNA() function to set up the application's global mapping.

The Mfile Application API

The **cmnMfileSDK.h** file defines prototypes for the Mfile application functions, which are implemented in the static library **libcmnMfile.a**.

setupGPNA()

The setupGPNA() function associates a GPNA memory-mapped file with an identifier that the service uses to reference it. Call this function only once from the function that performs the query.

This function has the following syntax and parameters:

```
void setupGPNA(int id, char *app_name, char *mfilename)
```

Parameters

- `id` is the identifier for the Mfile.
- `app_name` is the application name (NULL => GENERIC)
- m`filename` is the Mfile filename, including the full path

genericGPNA()

The genericGPNA() function allows you to query the memory-mapped file. It returns the entry that has the longest number of matching digits against the digits for the given nature.

This function has the following syntax and parameters and return value:

```
int genericGPNA(unsigned long memtableID, unsigned char nature, char *number,
                unsigned long start, unsigned long_minmatch,
                unsigned long maxmatch, GPNAresult_t *result, void **userdata,
                unsigned long *nummatched)
```

Parameters

- `memtableID` is the memory table ID, the index of the memory table as defined in the call to setupGPNA(). Value can be from 1 to `MAX_GPNA_MEM_TABLES`.
- `nature` is the nature of number (225=>no nature). Identifies a specific nature for the following number. For example, you can use this to specify the NOA of a number. In cases where raw digits should be analyzed, a pseudo nature (NULL = 0) should be used. Values can be from 0 to 254.
- `number` is a pointer to the number to find (NULL=>pointer to first word of singleEntry is returned in userdata). Maximum length is 28.
- `start` is the starting digit of that number from which the search will begin. Values can be from 1 to 28.
- `minmatch` is the minimum number of digits to match. A value of 0 indicates there is no minimum. Values can be from 0 to 28.
- `maxmatch` is the maximum number of digits to match. A value of 0 indicates there is no maximum. Values can be from 0 to 28.

- `result` is a pointer to the `GPNAResult_t` value. `GPNAResult_t` is the result of the genericGPNA() function.

  `userdata` is a pointer to the indexed value. Returns a pointer to the user-defined Mfile data associated with the search number.

- `nummatched` is the number of digits matched. A value of 0 indicates that no match was found. Values can be from 0 to 28.

Return

- 0 if successful
- -1 if a system or setup error occurred
- -2 if input parameters are invalid; for example, if maxmatch is less than minmatch.
- -3 if no entry is found that matches the set criteria

An Mfile Application Example

You can find an Mfile application example in the following SDK file:

$NCC_SDK_HOME/example/sdkMfileDaemon/sdkMfileAPI.cc

9

Creating and Replicating Database Tables

This chapter describes how to create and replicate database tables using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About Creating and Replicating Database Tables

The NCC SDK enables you to create database tables on the Service Management System (SMS) and replicate them to Service Logic Controllers (SLCs) for use in custom feature nodes. You could define tables, for example, to store information such as routing data that is specific to the topology of your network, allowing you to tailor services based on that information.

Creating and replicating custom database tables consists of the following tasks, which are described in the sections that follow:

1. Defining the database table.
2. Running the database table installer, **cmnTableInstaller.sh**, to create the database table and configure replication for it.

Defining a Database Table

You define your database table by creating an XML file using the XML elements shown in Table 9–1:

Table 9–1 Table Definition XML Elements

XML Element	Description
<TableDefinition>	Specifies the table name, as well replication and auditing information.
<TableColumnData>	Specifies the definition of a column in the table, including the column name, data type, and so on.
<TableConstraint>	Specifies constraints on the table such as what action to take when an event occurs on a column that references a column in another table.
<IndexDefinition>	Specifies the name and type of an index on the table.
<IndexColumnData>	Specifies the name of the column on which to create the index.

Each of these tags has attributes, which the following sections describe.

The TableDefinition Element

The <TableDefinition> element has the following format:

```
<TableDefinition TableName="MMX_ROUTING_SCHEME" RepBaseName="MMX_ROUTING_SCHEME"
RepPkgName="MMX_REP_INTERNAL" AuditBaseName="MMX_ROUTING_SCHEME" Comment="Models a
Routing Scheme" >
```

Table 9–2 describes the attributes of the <TableDefinition> element:

Table 9–2 TableDefinition Attributes

Attribute	Value
TableName	A string that contains the name of the table.
RepBaseName	A string that specifies the replication base name. It's used to create the names of the triggers that invoke the replication process. Four triggers are created with the following names: REP_<RepBaseName>_AIU REP_<RepBaseName>_AD REP_<RepBaseName>_BFR REP_<RepBaseName>_UPK
RepPkgName	A string that specifies the name of replication package. Leave this as the default, which is SMS.REP_INTERNAL.
AuditBaseName	A string that specifies the base name of the audit log. Determines the names of the triggers that invoke the audit process. Two triggers are created with names: <AuditBaseName>_ABT <AuditBaseName>_AAT If the AuditBaseName is empty, no audit triggers are created and changes to the table are not audited.
Comment	A string that provides an explanatory comment about the table, such as its purpose.

The TableColumnData Element

The <TableColumnData> element tag has the following format:

```
<TableColumnData ColumnName="ID" ReplicKey="T" Audited="T" ColumnDataType="NUMBER"
ColumnLength="28" ColumnPrecision="28" ColumnScale="0" AllowNull="F"
SeqBaseName="MMX_ROUTING_SCHEME" SeqMax="nomaxvalue" Comment="Primary key" />
```

Table 9–3 describes the attributes of the <TableColumnData> element:

Table 9–3 TableColumnData Attributes

Attribute	Value
ColumnName	A string that specifies the name of the column.
ReplicKey	A value of T for true or F for false that indicates whether this column forms part of the key for replication.
Audited	A value of T for true or F for false that indicates whether changes to this column are included in the audit log.
ColumnDataType	A string that specifies the column's data type.
ColumnLength	A string that specifies the maximum length of the column
ColumnPrecision	A string that specifies the decimal position for numeric values

Table 9–3 (Cont.) TableColumnData Attributes

Attribute	Value
ColumnScale	A string that specifies the maximum number of digits to the right of the decimal place
AllowNull	A value of T for true or F for false that indicates whether the column can have a value of NULL
SeqBaseName	A string that defines the name of a sequence to populate this column. A sequence called <SeqBaseName>_SEQ will be created by the installer and used by a trigger called <SeqBaseName>_MT.
SeqMax	A string that indicates sequence maximum.
Comment	A string that describes the column.

The TableConstraint Element

The <TableConstraint> element has the following format:

```
<TableConstraint ConstraintType="P" State="ENABLED" ReferencedSchema=""
ReferencedTable="" CascadeOnDelete="" CheckCondition="" ConstraintName="RT_SC_PK">
```

Table 9–4 describes the attributes of the <TableConstraint> element:

Table 9–4 TableConstraint Attributes

Attribute	Value
ConstraintType	A string that specifies the type of constraint in force. Possible values are: P - Primary key U - Unique R - Reference (foreign key) C - Check (uses the CheckCondition element)
State	A string that has a value of ENABLED to indicate that the constraint is in force.
ReferencedSchema	A string that specifies the name of the schema that the constraint references. This element applies only to a constraint type of R.
ReferencedTable	A string that specifies the name of the table that the constraint references. This element applies only to a constraint type of R.
CascadeOnDelete	A string that specifies the cascading action to take when a delete occurs on a row in the table. A value of CASCADE specifies that associated rows in a child table should be deleted when a row in this table is deleted. A value of NO ACTION indicates no action will be taken. This element applies only to a constraint type of R.
CheckCondition	A string that indicates a particular condition exists.

The IndexDefinition Element

The <IndexDefinition> element has the following format:

```
<IndexDefinition IndexName="RT_SC_PK" Unique="T">
```

Table 9–5 describes the attributes of the <IndexDefinition> element:

Table 9–5 IndexDefinition Attributes

Attribute	Value
IndexName	A string that specifies the name of the index.
Unique	A value of T for true or F for false that indicates whether the index is unique.

The IndexColumnData Element

The <IndexColumnData> element has the following format:

```
<IndexColumnData ColumnName="ID" />
```

Table 9–6 describes the attribute of the <IndexColumnData> element:

Table 9–6 IndexColumnData Attribute

Attribute	Value
ColumnName	A string that specifies that name of the column that is being indexed.

A Table Definition Example

You can find several instances of XML table definitions in the following SDK files:

$NCC_SDK_HOME/example/ABC/db/SDK.xml

$NCC_SDK_HOME/example/ABC/db/SDK_Client.xml

The following example shows the XML elements and attributes that define the SDK_ABC_NOA database table in the **SDK.xml** file:

```
    <TableDefinition TableName="SDK_ABC_NOA" RepBaseName="SDK_ABC_NOA" AuditBase
Name="SDK_ABC_NOA" Comment="Sample table for SDK PI commands, screens etc.">
        <TableColumnData ColumnName="NOA"         ReplicKey="T"  Audited="T" Col
umnDataType="NUMBER" ColumnLength="3" ColumnPrecision="3" ColumnScale="0" AllowN
ull="F" Comment="Primary key" />
        <TableColumnData ColumnName="DESCRIPTION" Replicated="T" Audited="T" Col
umnDataType="VARCHAR2" ColumnLength="50" ColumnPrecision="0"  ColumnScale="0" Al
lowNull="T" Comment="Optional free text" />
        <TableColumnData ColumnName="CHANGE_USER" ColumnDataType="CHAR"     Colu
mnLength="50" ColumnPrecision="0" ColumnScale="0" AllowNull="F" />
        <TableColumnData ColumnName="CHANGE_DATE" ColumnDataType="DATE"     Colu
mnLength="7"  ColumnPrecision="0" ColumnScale="0" AllowNull="F" />
        <TableColumnData ColumnName="CHANGE_TERM" ColumnDataType="VARCHAR2" Colu
mnLength="12" ColumnPrecision="0" ColumnScale="0" AllowNull="F" />
        <TableColumnData ColumnName="CHANGE_REF"  ColumnDataType="VARCHAR2" Colu
mnLength="50" ColumnPrecision="0" ColumnScale="0" AllowNull="T" />

        <TableConstraint ConstraintType="P" State="ENABLED" ConstraintName="SDK_
ABC_NOA__PK" UseIndex="True">
            <ReferenceColumns TableColumn="NOA" />
        </TableConstraint>
    </TableDefinition>
```

Running the Database Table Installer

The database table installer, **cmnTableInstaller.sh**, creates the database table from the supplied XML file and configures replication for it. You can find the database installer in the following location:

$NCC_SDK_HOME/bin

The installer has the following command line options:

```
cmnTableInstaller.sh -U <ora_username> -D <OutputDir> -S <tableSchemaFile> -C
<tableClientFile> [-options]
```

Table 9–7 describes the command line options for cmnTableInstaller.sh:

Table 9–7 cmnTableInstaller.sh Command Line Options

Parameter	Description
ora_username	The name of the owner of the table to be installed.
OutputDir	The output directory for the script.
tableSchemaFile	The name of the XML file that contains the table's schema
tableClientFile	The name of an XML file that defines storage information for tables and indexes (sizing, tablespaces etc.). This is generally specific to a particular installation.
options:	
-t	Node type of SMS (default) or Other.
-p	Output prefix (describing the component being installed)
-n	New install - abort installation if any of the tables already exists
-v	Verbose mode for debugging output
-h	Print command line help message

Defining the tableClientFile

The tableClientFile is an XML file that specifies storage information, such as sizing, tablespaces, and so on, for tables and indexes that are generally specific to a particular installation.

You define the tableClientFile by creating an XML file using the XML elements shown in Table 9–8.

Table 9–8 tableClientFile XML Elements

Element	Description
<ClientTableDefinition>	Specifies the storage attributes for a table
<ClientIndexDefinition>	Specifies the storage attributes for an index

The ClientTableDefinition Element

The <ClientTableDefinition element has the following format:

```
<ClientTableDefinition PCTINCREASE="0" INITRANS="1" MAXEXTENTS="UNLIMITED" M
INEXTENTS="1"  NEXTEXTENT="1M" INITIAL="1M" BUFFER_POOL="KEEP" PCTFREE="10" PCTU
SED="80" TABLESPACE="SDK_DATA" TableName="SDK_ABC_NOA" MAXTRANS="255" CACHE="Fal
se" />
```

Table 9–9 describes the attributes of the <ClientTableDefinition> element:

Table 9–9 ClientTableDefinition Attributes

Attribute	Value
PCTINCREASE	In locally managed tablespaces, Oracle Database uses PCTINCREASE to determine the initial segment size during segment creation. It ignores the parameter during subsequent space allocation. In dictionary-managed tablespaces, specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, which means that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system.
INITRANS	Specifies the initial number of concurrent transaction entries that will be allocated within each data block that is allocated to the database object. Value can range from 1 to 255 and defaults to 1, or, for a cluster, 2 or the default INITRANS value of the tablespace in which the cluster resides, whichever is greater.
MAXEXTENTS	For objects in dictionary-managed tablespaces, specifies the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1; rollback segments have a minimum of 2. Default value depends on your data block size. Oracle ignores MAXEXTENTS for objects in a locally managed tablespace. Specify UNLIMITED to automatically allocate extents as needed. Oracle recommends UNLIMITE to minimize fragmentation.
MINEXTENTS	In a locally managed tablespace, Oracle uses MINEXTENTS to compute the initial amount of space to allocate, which is equal to INITIAL * MINEXTENTS. Subsequently, Oracle sets the value to 1. In a dictionary-managed tablespace, MINEXTENTS is the minimum number of extents that must be allocated to the segment. In dictionary-managed tablespaces, specify the total number of extents to allocate when the object is created. The minimum and default value is 1, in which case Oracle allocates only the initial extent. For rollback segments, the minimum and default value is 2. The maximum value depends on your operating system.
NEXTEXTENTS	Specifies the size in bytes of the next extent to allocate to the object. In locally managed tablespaces, if the tablespace is set for automatically allocate extent management Oracle determines the size. In UNIFORM tablespaces, the size of NEXTEXTENT is the uniform extent size specified when the tablespace was created. In a dictionary-managed tablespace, the default value is the size of 5 data blocks. Minimum value is the size of 1 data block. Maximum value depends on your operating system. For values less than 5 data blocks, Oracle rounds the value up to the next multiple of the data block size. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation.
INITIAL	Specify the size of the first extent of the object. Oracle allocates space for this extent when you create the schema object. In locally managed tablespaces, the value of INITIAL, in conjunction with the values of MINEXTENTS, NEXT and PCTINCREASE, determines the initial size of the segment
BUFFER_POOL	Lets you specify a default buffer pool for a schema object. All blocks for the object are stored in the specified cache.

Table 9–9 (Cont.) ClientTableDefinition Attributes

Attribute	Value
PCTFREE	A number that represents the percentage of space in each data block of the database object that Oracle reserves for future updates to the object. Specified as a value from 0 to 99. A value of 0 means that the entire block can be filled by inserts of new rows. Defaults to 10. Total of PCTUSED and PCTFREE cannot be greater than 100.
PCTUSED	A number that represents the minimum percentage of used space that Oracle maintains for each data block of the database object. Specified as a positive integer from 0 to 99 and defaults to 40. Total of PCTUSED and PCTFREE cannot be greater than 100.
TABLESPACE	Specifies the name of the tablespace in which Oracle Database will create the table. Tablespace represents an allocation of space in the database to store schema objects. See the SQL CREATE TABLESPACE statement for more information.
TableName	Specifies the name of the database table to which this definition applies.
MAXTRANS	Determines the maximum number of concurrent update transactions allowed for each data block in the segment.
CACHE	When a full table scan is performed, CACHE specifies that you want the blocks retrieved for this cluster to be placed at the most recently used end of the least recently used (LRU) list in the buffer cache. Useful for small lookup tables.

The ClientIndexDefinition Element

The <ClientIndexDefinition> element has the following format:

```
<ClientIndexDefinition IndexName="SDK_ABC_EXAMPLE__PK" TABLESPACE="SDK_DATA"
 INITIAL="1M" NEXTEXTENT="1M" MINEXTENTS="1" MAXEXTENTS="UNLIMITED" BUFFER_POOL=
"KEEP" PCTFREE="10" PCTINCREASE="0" INITRANS="2" MAXTRANS="255" />
```

Table 9–10 describes the attributes of the <ClientIndexDefinition> element:

Table 9–10 ClientIndexDefinition Attributes

Attribute	Value
IndexName	Specifies the name of the index.
TABLESPACE	Specifies the name of the tablespace in which Oracle Database will create the table. Tablespace represents an allocation of space in the database to store schema objects. See the SQL CREATE TABLESPACE statement for more information.
INITIAL	Specify the size of the first extent of the object. Oracle allocates space for this extent when you create the schema object. In locally managed tablespaces, the value of INITIAL, in conjunction with the values of MINEXTENTS, NEXT and PCTINCREASE, determines the initial size of the segment

Table 9–10 (Cont.) ClientIndexDefinition Attributes

Attribute	Value
NEXTEXTENT	Specifies the size in bytes of the next extent to allocate to the object. In locally managed tablespaces, if the tablespace is set for automatically allocate extent managementOracle determines the size. In UNIFORM tablespaces, the size of NEXTEXTENT is the uniform extent size specified when the tablespace was created. In a dictionary-managed tablespace, the default value is the size of 5 data blocks. Minimum value is the size of 1 data block. Maximum value depends on your operating system. For values less than 5 data blocks, Oracle rounds the value up to the next multiple of the data block size. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation.
MINEXTENTS	In dictionary-managed tablespaces, specify the total number of extents to allocate when the object is created. The minimum and default value is 1, in which case Oracle allocates only the initial extent. For rollback segments, the minimum and default value is 2. The maximum value depends on your operating system. In a locally managed tablespace, Oracle uses MINEXTENTS to compute the initial amount of space to allocate, which is equal to INITIAL * MINEXTENTS. Subsequently, Oracle sets the value to 1. In a dictionary-managed tablespace, MINEXTENTS is the minimum number of extents that must be allocated to the segment.
MAXEXTENTS	For objects in dictionary-managed tablespaces, specifies the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1; rollback segments have a minimum of 2. Default value depends on your data block size. Oracle ignores MAXEXTENTS for objects in a locally managed tablespace. Specify UNLIMITED to automatically allocate extents as needed. Oracle recommends UNLIMITE to minimize fragmentation.
BUFFER_POOL	Lets you specify a default buffer pool for a schema object. All blocks for the object are stored in the specified cache.
PCTFREE	A number that represents the percentage of space in each index block that Oracle reserves for future updates to the object. Specified as a value from 0 to 99. A value of 0 means that the entire block can be filled by inserts. Defaults to 10.
PCTINCREASE	In locally managed tablespaces, Oracle Database uses PCTINCREASE to determine the initial segment size during segment creation. It ignores the parameter during subsequent space allocation. In dictionary-managed tablespaces, specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, which means that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system.
INITRANS	Specifies the initial number of concurrent transaction entries that will be allocated within each data block that is allocated to the index. Value can range from 1 to 255 and defaults to 2.
MAXTRANS	Determines the maximum number of concurrent update transactions allowed for each data block in the segment.

A tableClientFile Example

The following example shows a sample definition of a tableClientFile file.

```
<?xml version="1.0"?>
<!DOCTYPE client SYSTEM "../../../bin/client.dtd">
<client>

    <ClientTableDefinition PCTINCREASE="0" INITRANS="1" MAXEXTENTS="UNLIMITED" M
INEXTENTS="1"  NEXTEXTENT="1M" INITIAL="1M" BUFFER_POOL="KEEP" PCTFREE="10" PCTU
SED="80" TABLESPACE="SDK_DATA" TableName="SDK_ABC_NOA" MAXTRANS="255" CACHE="Fal
se" />
    <ClientTableDefinition PCTINCREASE="0" INITRANS="1" MAXEXTENTS="UNLIMITED" M
INEXTENTS="1"  NEXTEXTENT="1M" INITIAL="1M" BUFFER_POOL="KEEP" PCTFREE="10" PCTU
SED="80" TABLESPACE="SDK_DATA" TableName="SDK_ABC_NUMBER_BARRED" MAXTRANS="255"
CACHE="False" />
    <ClientTableDefinition PCTINCREASE="0" INITRANS="1" MAXEXTENTS="UNLIMITED" M
INEXTENTS="1"  NEXTEXTENT="1M" INITIAL="1M" BUFFER_POOL="KEEP" PCTFREE="10" PCTU
SED="80" TABLESPACE="SDK_DATA" TableName="SDK_ABC_EXAMPLE" MAXTRANS="255" CACHE=
"False" />

    <ClientIndexDefinition IndexName="SDK_ABC_EXAMPLE__PK" TABLESPACE="SDK_DATA"
 INITIAL="1M" NEXTEXTENT="1M" MINEXTENTS="1" MAXEXTENTS="UNLIMITED" BUFFER_POOL=
"KEEP" PCTFREE="10" PCTINCREASE="0" INITRANS="2" MAXTRANS="255" />
    <ClientIndexDefinition IndexName="SDK_ABC_NOA__PK" TABLESPACE="SDK_DATA" INI
TIAL="1M" NEXTEXTENT="1M" MINEXTENTS="1" MAXEXTENTS="UNLIMITED" BUFFER_POOL="KEE
P" PCTFREE="10" PCTINCREASE="0" INITRANS="2" MAXTRANS="255" />

</client>
```

Replicating Tables

When you run the database table installer, you register all of the tables with SMS. You can then select which elements you want replicated, and to which nodes, through the standard SMS Node Management screen. For more information, see the *Service Management System User's Guide*.

To replicate tables, you must include the XML elements for replication in the table schema file (tableSchemaFile parameter) that you submit to the database installer. The replication section of the XML uses the elements shown in Table 9–11.

Table 9–11 XML Elements for Replication

XML Element	Description
<Replication>	Specifies the application name and the application ID.
<Platforms>	Contains the <Platform> elements.
<Platform>	Specifies the names of the platforms on which the tables for this application will be replicated.
<Groups>	Contains <TableReference> and <Group> elements.
<TableReference>	Specifies the name of a table defined in a <TableDefinition> element.
<Group>	Specifies the name of a group of tables to be replicated. You can specify </Group> following a <TableReference> element to create a default group with the same name as the table.
<Dependency>	Specifies a dependency relationship between tables to control the order in which data is replicated.

You can find additional information about these elements in the following file:

$NCC_SDK_HOME/bin/schema.dtd

The Replication Element

The <Replication> element has the following format:

```
<Replication ApplicationID="901" ApplicationName="ABC" Description="Test Install
Table" Status="4" DisplayName="TEST" Version="1.0">
```

Table 9–12 describes the attributes of the <Replication> element:

Table 9–12 Replication Attributes

Attribute	Description
ApplicationID	Specifies the application ID. Custom applications use IDs in the range of 900-999.
ApplicationName	A string that specifies the name of the application.
Description	A description of the replication.
Status	A string that specifies the status.
DisplayName	The name to display in screens.
Version	A string that specifies the version.

The Platforms Element

The <Platforms> element contains <Platform> elements and has no attributes. It has the following format.

```
<Platforms>
   <Platform Type="SLC"></Platform>
</Platforms>
```

The Platform Element

The <Platform> Element is used to specify the types of platforms to which the tables for this application need to be replicated. It has the following format:

```
<Platform Type="SLC"></Platform>
```

The <Platform> element has only the Type attribute, which is a string that specifies the type of platform to which the tables needs to be replicated.

The Groups Element

The <Groups> element contains <Group> elements and has no attributes. It has the following format:

```
<Groups>
   <TableReference Name="TABLE1">
      <Group/>
   ...
</Groups>
```

The Group Element

The <Group> element allows you to group tables for replication and to reference a group of tables when defining dependency relationships. It has the following format:

```
<Group Name="SDK_ABC_MAIN" />
```

The <Group> element has only the Name attribute, which is a string that specifies the name of the table group. If you don't include the Name attribute, as shown in the

following example, the name of the group defaults to the name of the table in the preceding <TableReference> element.

```
<Groups>
   <TableReference Name="TABLE1">
      <Group/>
```

Once you have defined groups, you can define dependency relationships for replication.

The Dependency Element

The <Dependency> element allows you to define dependency relationships to control the order in which data is replicated. The <Dependency> element has the following format:

```
<Dependency Dependent="Group1" DependsOn="Group2" />
```

Table 9–13 describes the attributes of the <Dependency> element:

Table 9–13 Dependency Attributes

Attribute	Description
Dependent	A string that specifies the name of a group (not a table) that has a dependency.
DependsOn	A string that specifies the name of a table on which the dependent group depends.

A Table Replication Example

The following example illustrates a table replication definition:

```
    <Replication ApplicationID="901" ApplicationName="ABC" Description="Test Ins
tall Table" Status="4" DisplayName="TEST" Version="1.0">
        <Platforms>
            <Platform Type="SLC"></Platform>
            <Platform Type="VWS"></Platform>
        </Platforms>

        <Groups>
            <TableReference Name="SDK_ABC_NOA">
                <!-- The group name defaults to the table name -->
                <Group />
            </TableReference>
            <TableReference Name="SDK_ABC_NUMBER_BARRED">
                <!-- You can also specify the group name explicitly -->
                <Group Name="SDK_ABC_MAIN" />
            </TableReference>
            <TableReference Name="SDK_ABC_EXAMPLE">
                <!-- And groups can be replicated without being part of a
                     dependency relationship -->
                <Group />
            </TableReference>
        </Groups>

        <Dependency Dependent="SDK_ABC_MAIN" DependsOn="SDK_ABC_NOA" />
        <!-- A new replication group can also depend on an existing (e.g. produc
t) group -->
        <Dependency Dependent="SDK_ABC_MAIN" DependsOn="ACS_CUSTOMER" />
    </Replication>
```

10

Creating an EDR Loader Plugin

This chapter describes how to create Event Detail Record (EDR) loader plugins using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About EDR Loader Plugins

The Voucher and Wallet Server produces an EDR for every change to a wallet or voucher. An EDR loader process on the SMS processes each EDR. The SDK includes an API that allows you to create a custom EDR loader plugin.

> **Note:** Some of the EDR function names and file names still refer to CDR (Call Detail Record) for legacy reasons, but they are actually EDRs and not specific to calls.

A plugin could add more information to the EDR. For example, if an EDR contains a numeric ID, you might write a plugin that replaces the ID with a more descriptive name. A plugin also might filter some EDRs that must be passed to an external billing system, for post-paid billing perhaps. In this case, the plugin would select the relevant EDRs based on the values in its fields, then write a new EDR in the format required by the external system.

The EDR Loader Plugin Shared Library

You must make all EDR loader plugins part of a shared library that is configured to be loaded by the EDR loader. When it starts up, the EDR Loader calls the shared library's initialization function, `ccsCDRLoaderPluginLibInit()`. You must implement this function and also register any plugins that are in the library using the `registerCDRLoaderPlugin()` function.

In the following example, `ccsCDRLoaderPluginLibInit()` registers two plugins:

```
extern "C" ccs::cdr::CDRLoaderPluginLib *ccsCDRLoaderPluginLibInit() {
    // Register first plugin
    ccs::cdr::CDRLoaderPluginLib *lib =
ccs::cdr::registerCDRLoaderPlugin(myLibraryName, new FirstPlugin());

    // Register other plugins
    ccs::cdr::registerCDRLoaderPlugin(myLibraryName, new SecondPlugin());
    // ...

    return lib;
}
```

As with service loaders and macro nodes, you must place the shared library in a directory that you have specified in the LD_LIBRARY_PATH environment variable.

The ccsCDRLoader section of the **eserv.config** file lists the shared libraries that will be loaded. For more information about using the ccsCDRLoader, see the *Charging Control Services Technical Guide.*

The EDR Loader Plugin

An EDR loader plugin implements and derives from the `ccs::cdr::CDRLoaderSDKPlugin` class and must implement the `process()` and `flush()` functions.

The EDR loader calls the `process()` function for every EDR and `flush()` when the whole file has been processed.

In the following example, the process() function identifies the EDR that it's currently processing, totals costs, if present, and records the processing time.

```
bool sdkCDRLoaderPlugin::process(ccs::cdr::CDRInterface &cdr) {
    DOUT << "Plugin " << getName() << " processing CDR with sequence number " <<
 cdr.getSeqNum() << std::endl;

    // ACCOUNT_TYPE=1|ACCT_ID=4|ACCT_REF_ID=4|ACS_CUST_ID=1|BALANCES=66076|BALAN
CE_TYPES=9|
    // BILLING_ENGINE_ID=1|CDR_TYPE=9|CLI=094454600|COSTS=-3000|CS=S|NEW_BALANCE
_EXPIRIES=|
    // OLD_BALANCE_EXPIRIES=|RECORD_DATE=20110328185229|RESULT=Success|SCP_ID=12
4153970|
    // SEQUENCE_NUMBER=2365233167|TERMINAL=192.168.10.229|USER=SU|WALLET_TYPE=3

    if (cdr.hasTag("COSTS")) {
        // Record the consumption of costs and update our running total
        const std::string &costs = cdr.getValue("COSTS");
        DOUT << "Has some costs: " << costs << std::endl;
    }

    // Record the fact that we were processed by this plugin, and when
    cdr.addExtraField("SDK","Y");
    time_t now = time(NULL);
    cdr.addExtraField("SDK_WHEN", ccs::cdr::CDRInterface::getGMTTimeString(now))
;

    return true;
}
```

Here the flush() function simply records that it is being called and returns a value of true:

```
bool sdkCDRLoaderPlugin::flush() {
    DOUT << "Plugin " << getName() << " is being flushed" << std::endl;
    return true;
}
```

You can find the complete example in the file in the **$NCC_SDK_HOME/example/sdkCDRLoaderPlugin/sdkCDRLoaderPlugin.cc** file.

11

Creating a CcsAuth Voucher PAM Plugin

This chapter describes how to create a custom CcsAuth Voucher PAM plugin using the Oracle Communications Network Charging and Control (NCC) Software Development Kit (SDK).

About CcsAuth PAM Plugins

Generating a voucher secret HRN and a hashed/encrypted version of the HRN, and then validating an input HRN is handled by CcsAuth PAM plugin libraries in CCS. The SDK includes an API that allows you to create a custom CcsAuth PAM plugin.

With a custom PAM plugin, you can define custom solutions for the following actions:

- generating an HRN (secret) from voucher number and seed information.
- encrypting a secret to produce a private secret.
- decrypting a private secret to recover the HRN (secret).

A plugin can change how the HRN is generated. Five inputs are available to use when generating a custom HRN: the voucher number, a sequence derived from the CB10 HRN, and three integer parameters. One possible custom HRN might be to generate an alphanumeric HRN instead of the numeric HRN created by CB10. Apart from HRN generation, a plugin can also specify its own custom method for encrypting/hashing the secret (HRN) to produce the private secret which is the entity stored in the database against the voucher.

You can also specify whether the private secret so produced is decryptable (as would be the case if the method that produced the private secret used symmetric encryption instead of one-way hashing).

You can define and install up to three different custom voucher PAM plugins with the SDK.

Once a custom SDK plugin is installed it is available as a selection from the PAM drop-down menu when creating new CCS Authentication Rules in the Voucher Security screen. Installed SDK plugins also have their own button for key generation in the Voucher Security Screen.

A SDK voucher PAM plugin is installed or uninstalled from the database on the SMS by using the provided **ccsAuthPluginInstaller** command line utility, which takes an input XML directive file describing the SDK voucher PAM plugin to be installed or uninstalled.

The CcsAuth Plugin Shared Libraries

Each defined SDK voucher PAM plugin is a different shared library. The shared library is created by sub classing a SDK delivered abstract class defined by header file ccsAuthPluginSDK.hh, providing definitions for pure virtual methods and overriding virtual methods with default implementations as required. When the shared library is loaded by the Ccs Auth subsystem, a factory routine is called to create an instance of the derived class that the library defines. Similarly a factory routine must be defined to delete an instance of the derived class.

If for example the plugin derived class is MyVoucherAuth, then these two routines need to be implemented globally and declared extern "C" in the plugin class:

```
extern "C" ccs::auth::ccsAuthPluginSDK* createPluginInstance() {
   return new MyVoucherAuth;
}
extern "C" void destroyPluginInstance(ccs::auth::ccsAuthPluginSDK* p) {
   delete p;
}
```

As with service loaders and macro nodes, you must place the shared library in a directory that you have specified in the LD_LIBRARY_PATH environment variable.

SDK Voucher PAM plugin shared libraries must be installed on all NCC nodes (SMS, SLC, and VWS).

SDK Voucher PAM plugins

A voucher PAM plugin implements and derives from the ccs::auth::CcsAuthPluginSDK abstract class and must implement these pure virtual methods from that class:

```
const bool usingCB10ForSecret()
const bool usingSHA256ForPrivateSecret()
const bool usingSHA512ForPrivateSecret()
const bool usingAES256ForPrivateSecret()
```

Method **usingCB10ForSecret()** should be defined to return true if the CB10 algorithm only is being used to create the secret (HRN) for a voucher. If a custom secret creation is required then define usingCB10ForSecret() to return false and provide a definition of the custom secret method by redefining virtual method makeSecret (see below).

The three **using*ForPrivateSecret()** methods are used to define which product supported method of producing the private secret (encrypted/hashed HRN) from the secret is required. To use a specific product method, e.g. SHA512, define the corresponding routine (usingSHA512ForPrivateSecret()) to return true and define the other two using*ForPrivateSecret() methods to return false. To use a custom method for creating the private secret, define all three using*ForPrivateSecret() methods to return false and then provide an implementation for method makePrivateSecret().

These following methods are virtual in ccs::auth::CcsAuthPluginSDK and may be overridden if desired in the SDK subclass:

```
int makeSecret(const std::string& key, const int sdk_p1, const int sdk_p2, const
int sdk_p3, std::string& secret)
bool makePrivateSecret(const std::string& secret, std::string& private_secret)
bool canDecryptPrivateSecret()
bool regenerateSecretFromPrivateSecret()
```

These methods are called as required by the application framework.

Method **makeSecret** is called by the application framework if usingCB10ForSecret() is defined to return false.

These arguments to **makeSecret** are then available to produce the secret which should be returned in the final argument parameter, std::string& secret.

The **makeSecret** input parameters are:

`const std::string& key` - the input key (voucher number).

`const std::seed_seq& seedSeq` - a seed sequence initialized with the CB10 produced HRN.

`const int sdk_s_length` - the SDK Custom Secret Length value specified by the Authentication Rule being used. This value allows the length of the custom HRN (in characters) to be specified. This feature is used if the plugin is installed on the SMS with the CustomHRN value specified as Y in the specification XML file for the plugin See section "ccsAuthPluginInstaller".

```
const int sdk_p1
```

```
const int sdk_p2
```

```
const int sdk_p3
```

These values can be optionally defined in the Authentication Rule screen (fields SDK P1, SDK P2, and SDK P3) and can be used as part of the **makeSecret** implementation. If they are left blank in the Authentication Rule screen, then their values are 0 in the **makeSecret** call.

The output parameter for **makeSecret** is

```
std::string& secret
```

which should contain the created secret string of length sdk_s_length characters.

The return value of method **makeSecret** is int.

If **makeSecret** is successful, return the value 1 from the method. If there is an error, return the value 0.

Method **makePrivateSecret** is called when all using*ForPrivateSecret() methods are defined as returning false. In this case, the implementation for creating a private secret from an input secret (HRN) should be supplied by redefining **makePrivateSecret**.

The input parameters for **makePrivateSecret** are secret and key.

`const std::string& secret` - the input sequence (HRN) to produce a private secret from.

`const std::string& key` - a custom encryption key generated for this plugin. This is a random string of 256 hexadecimal characters created when the Generate button for the plugin is used on the **Service Management > Security > Voucher Security** panel.

The output parameter for **makePrivateSecret** is private_secret.

`std::string& private_secret` - should contain the produced private secret as a hexadecimal string.

The return value from **makePrivateSecret** is bool and should be true if the private secret was created successfully and false otherwise.

Method **canDecryptPrivateSecret** is used to indicate whether the chosen private secret production method uses a scheme that can be reversed to recover the secret from the private secret. This will be possible if a form of symmetric encryption was used to

produce the private secret, but will not be possible if hashing is used as that is a one-way process only.

If **makePrivateSecret** is being used and the defined private secret production method is reversible, define **canDecryptPrivateSecret** to return true. (The default version return false).

Method **regenerateSecretFromPrivateSecret** is used to implement the "regeneration" method, the method that decrypts/reverses the private secret production method. This method is called by the application framework if **canDecryptPrivateSecret** returns true. **regenerateSecretFromPrivateSecret** takes as input private_secret and key.

`const std::string& private_secret` - private secret to be decrypted, hexadecimal string.

`const std::string& key` - custom encryption key for the plugin, 256 character hexadecimal string, same value is provided to **makePrivateSecret** by its key argument.

The output parameter is secret.

`std::string& secret` - should contain the decrypted secret string

regenerateSecretFromPrivateSecret should return a bool value, true if the secret was recovered successfully from the private_secret and false otherwise.

The following utility methods are also provided:

`const std::string& getName()` - returns the name of the plugin.

`static const size_t maxKeyLen()` - returns the maximum supported input key length (voucher number).

`static const size_t maxKeyLenZ()` - returns the maximum supported input key length (null terminated).

`static const maxSecretLen()` - maximum supported length of a secret (not null terminated).

`static const maxPrivateSecretLen()` - maximum supported length of a private secret (hexadecimal string representation), not null terminated.

`static const size_t maxSecretLenZ()` - maximum length of a secret (null terminated).

`static const size_t maxPrivateSecretLenZ()` - maximum length of a private secret (hexadecimal string representation), null terminated.

You can find the complete example in $NCC_SDK_HOME/example/sdkCcsAuthPlugin/sdkCcsAuthPlugin.cc and .hh files.

ccsAuthPluginInstaller

This utility installs and uninstalls a SDK Voucher PAM plugin from the database on the SMS. After a plugin has been developed, the definition of the plugin needs to be installed in the database so that:

- The Ccs Auth subsystem knows the plugin's name, its properties, and the path to the shared library implementing the plugin.
- The screens can display a Generate button for the plugin in the Voucher Security tab.
- The screens can populate the PAM drop-down menu with the name of the plugin as one of the available selections when defining a new Authentication Rule.

The ccsAuthPluginInstaller has the following usage:

```
ccsAuthPluginInstaller [-i | -u] -f specFile.xml
```

where -i says to install the plugin definition in the database and -u says to uninstall an existing plugin definition from the database. The plugin specification is supplied in a xml file named with the -f option.

Here is a specification xml file for the example sdkCcsAuthPlugin:

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

<!DOCTYPE CcsAuthPluginDefinitions SYSTEM "pluginschema.dtd"[]>

<CcsAuthPluginDefinitions>

<CcsAuthPlugin>

<DisplayName>SDK ccsAuthPlugin</DisplayName>

<LibraryPath>/IN/service_packages/CCS/lib/libsdkCcsAuthPlugin.so</LibraryPath>

<CustomHRN>Y</CustomHRN>

<UsesIterations>N</UsesIterations>

<Description>SDK ccsAuthPlugin example</Description>

<SupportsDecryption>N</SupportsDecryption>

</CcsAuthPlugin>

</CcsAuthPluginDefinitions>
```

An xml file contains one **<CcsAuthPluginDefinitions>** element that can contain one or more **<CcsAuthPlugin>** elements.

A **<CcsAuthPlugin>** should define these elements:

DisplayName - the name of the plugin as it presented to the user in the GUI on the label for a Generate button or as a selection in a PAM drop-down box.

LibraryPath - the absolute path name to the location of the plugin shared library. Should normally be a location in /IN/service_packages/CCS/lib.

CustomHRN - whether the plugin is implementing its own method for HRN generation as an adjuct to CB10. If Y, then the Authentication Rule field **SDK Custom Secret Length** becomes enabled.

UsesIterations - whether this plugin supports the specification of a number of iterations (> 1) for producing the private secret from the secret (if hashing is being used for example).

Description - a longer form description for the plugin.

SupportsDecryption - whether this plugin supports decrypting the private secret to recover the secret (HRN). This controls whether the decryption is attempted using the plugin, and whether GUI buttons to decrypt the HRNs are present for authorised screens users.