Oracle® Communications MetaSolv Solution

CORBA API Developer's Reference

Release 6.3

E69847-02

February 2017

ORACLE®

Oracle Communications MetaSolv Solution CORBA API Developer's Reference, Release 6.3

E69847-02

Contents

3 Common Architecture

4 The Infrastructure API

5 The Inventory and Capacity Management API

6 The Number Inventory API

7 The Activation API

8 The Plant API

9 The PSR Ancillary API

10 The PSR Order Entry API

11 The Switch Provisioning Activation API

12 The Transport Provisioning Activation API

13 The Trouble Management API

D The PSR End User Billing API

Glossary

Preface

This document accompanies the application programming interfaces (APIs) that form the Oracle Communications MetaSolv Solution (MSS) interface architecture. Information includes how the MSS APIs work, high-level information about each API, and instructions for using the MSS APIs to perform specific tasks.

Audience

This document is intended for CORBA developers who are developing applications that use the MSS APIs.

This document assumes you have a working knowledge of the Common Object Request Broker Architecture (CORBA) standards, including an understanding of interface definition language (IDL). For details about CORBA fundamentals and programming or IDL syntax, refer to the documentation for your CORBA implementation.

This document helps you develop:

- A high-level understanding of the general principles that govern the use of the APIs.
- An understanding of the details of implementing an application that uses a specific API.

This document is not intended as a training tool, nor does it address the installation or use of any Oracle products.To fully integrate the MSS APIs with your product often requires knowledge of a specific MSS functional area. For information about training or consulting services, contact your MSS representative.

This document contains:

- Key concepts
- Details about specific APIs
- Sample code that illustrates key concepts

Related Documents

For more information, see the following documents in the MSS documentation set:

- *MSS Planning Guide:* Describes information you need to consider in planning your MSS environment before installation.
- *MSS Installation Guide*: Describes system requirements and installation procedures for installing MSS.

- *MSS System Administrator's Guide:* Describes postinstallation tasks and administrative tasks such as maintaining user security.
- *MSS Security Guide:* Provides guidelines and recommendations for setting up MSS in a secure configuration.
- *MSS Database Change Reference:* Provides information on the database changes in MSS releases.
- *MSS Network Grooming User's Guide:* Provides information about the MSS Network Grooming tool.
- *MSS Address Correction Utility User's Guide:* Provides information about the MSS Address Correction utility.
- *MSS Technology Module Guide:* Describes each of the MSS technology modules.
- *MSS Data Selection Tool How-to Guide:* Provides an overview of the Data Selection Tool, and procedures on how it used to migrate the product catalog, equipment specifications, and provisioning plans from one release of your environment to another.
- *MSS Custom Extensions Developer's Reference:* Describes how to extend the MSS business logic with custom business logic through the use of custom extensions.
- *MSS Web Services Developer's Guide:* Describes the MSS Web Services and provides information about the MSS Web Service framework that supports web services, the various web services that are available, and how to migrate existing XML API interfaces to web service operations.

For step-by-step instructions for tasks you perform in MSS, log in to the application to see the online Help.

Documentation Accessibility

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

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

1

The MetaSolv Solution Architecture

This chapter provides a general understanding of the Oracle Communications MetaSolv Solution interface architecture, a group of APIs that allow access to the data in the MetaSolv Solution database. This chapter tells you how MetaSolv Solution provides access to information and software functionality to external applications.

What Does MetaSolv Solution Do?

Oracle is a leading provider of operations support system (OSS) solutions and professional services for service providers in the local exchange, interexchange, wireless, data, and Internet markets. MetaSolv Solution enables service providers to automate and manage their ordering, service activation, and service assurance (trouble management) processes.

The MetaSolv Solution product line is set of subsystems integrated by a common repository, a database, of business data and processes. Each subsystem supports a critical aspect of the service provider's business:

- Order Management: Enables the service provider to manage the end-to-end service delivery process. This often involves more than one type of service request or transaction within the organization, as well as transactions with other service or network providers.
- Service Provisioning: Facilitates delivery of a full spectrum of services, from simple circuit assignments to complex circuit design and configuration. The integration of the Service Provisioning subsystem with other MetaSolv Solution subsystems provides the service provider with an accurate view of what their customer ordered and what their network can support.
- Network Design: Brings together the geographical, physical, electrical, and logical dimensions of the network into a single, cohesive view supported by a set of integrated equipment administration and network design modules. This subsystem supports the design of networks and fulfillment of services across multiple providers and technologies.
- Trouble Management: Supports the reporting, tracking, and resolution of trouble associated with providing telecommunication products and services. This subsystem tracks a reported problem from its initial identification to its resolution.
- Interface Management: Supports accurate, reliable, and timely exchange of information between MetaSolv Solution and the service provider's other systems and external organizations.
- Work Management: Enables work to flow electronically across the organization. This subsystem provides the capability to manage provisioning plans, which are

groups of tasks needed to manage the flow of work and information required for the service fulfillment process.

MetaSolv Solution provides a flexible and open architecture. MetaSolv Solution uses an integrated database, where all data is collected and shared across all MetaSolv Solution subsystems. This single database provides a high-level information about the customer's profile.

How Do the MetaSolv Solution APIs Work with MetaSolv Solution?

MetaSolv Solution is developed on an open architecture that recognizes the need to electronically exchange information with a wide array of systems such as the managed network, other enterprise systems, and external trading or service partners and their operations support systems. The MetaSolv Solution APIs were to permit a flow of data between the MetaSolv Solution database and external applications.

Automatic and manual export event triggers exist within MetaSolv Solution providing end users with the capability to send work to the MetaSolv Solution Application Server or to third-party gateways.

The MetaSolv Solution interface architecture provides APIs that enable access to specific parts of data in the MetaSolv Solution database.

Table 1–1 describes the APIs that are available in MetaSolv Solution.

Table 1–1 MetaSolv Solution APIs

API	Description
End User Billing API	The End User Billing API publishes information needed for exporting data from the MetaSolv Solution database to support end-user billing from PSRs. This API integrates MetaSolv Solution order management and provisioning information with billing solutions, defines a standard end-user billing interface, and allows generic support for any billing vendor.
Inventory and Capacity Management (ICM) API	The ICM API provides beginning-to-end visibility of service and network assets, including facilities, equipment, and circuits.
Product Service Request (PSR) Ancillary API	The PSR Ancillary API permits exposure of E911 and LIDB/CNAM information to database providers.
Product Service Request (PSR) Order Entry API	The PSR Order Entry API enables a customer or a customer's third-party developer to insert customer account, service location, and PSR order information into the MetaSolv Solution database. This information is necessary for telecommunication products or services to be provisioned through MetaSolv Solution.
Switch Provisioning Activation API	The Switch Provisioning Activation API provides a vendor-independent interface that enables the flow-through provisioning of switch services such as POTS.
Transport Provisioning Activation API	The Transport Provisioning Activation API provides a vendor-independent interface that enables the flow-through provisioning of Frame Relay, ATM circuits, xDSL, and SONET. Transport provisioning reduces service turn-up time, staffing needs, and provisioning errors.
Trouble Management API	The Trouble Management API provides a mechanism for integrating the MetaSolv Solution Trouble Management subsystem with a third-party network or fault management system. This integration allows for the automatic creation of trouble tickets in the MetaSolv Solution Trouble Management subsystem from the fault management system.

Table 1–1 (Cont.) MetaSolv Solution APIs

API	Description
Work Management API	The Work Management API enables customers to generate tasks for an order, view tasks in work queues, and complete, reopen, transfer, or suspend tasks for an order through a web interface.
Infrastructure API	The Infrastructure API provides operations for exporting lists of information from the MetaSolv Solution database. The Infrastructure API can export these types of information from the database: ■ Structure formats and structure format components ■ Geographic areas and types ■ Code categories and code category values, including languages ■ Network locations
Number Inventory API	The Number Inventory API was created to more efficiently handle the administration of telephone numbers and inventory items in MetaSolv Solution. Operations are provided in the WDINI.IDL that provide the following functionality: ■ Export Number Inventory ■ Import Number Inventory ■ Generate User ID ■ Generate User Password ■ Validate Password ■ Update Number Inventory Provisioning ■ Pre-assign Telephone Numbers ■ Remove Inventory Association The following operations provide lookup and export functionality: ■ exportTopLevelDomains ■ exportInventoryTypes ■ exportInventorySubTypes ■ exportInventoryStatus ■ exportInventoryRelationTypes ■ exportInventoryItem ■ exportInventoryItems ■ exportInventoryItemAssociation ■ exportTelephoneNumbers ■ exportAccessTelephoneNumbers The following operations provide import functionality: ■ importNewInventoryItem ■ importUpdatedInventoryItem ■ importInventoryAssociation

Overview of Essential Terminology

Some terminology in the software and telecommunications industries differs from one provider to another. To eliminate confusion, this guide includes a glossary. However; as you read the remainder of this documentation, it is important that you understand the distinctions between the following sets of terms:

Solicited Messages Vs. Unsolicited Messages

The point of reference for this guide is the MetaSolv Solution product line. Therefore, when reading material about messages, whether the API is the initiator of the request determines whether a message is solicited or unsolicited. When MetaSolv Solution initiates the request and your software receives, that request is a solicited message. When your application initiates the request and the API receives the request, that request is called an unsolicited message.

Where the documentation does not refer specifically to solicited or unsolicited messages, the information applies to both solicited and unsolicited messages.

Events Vs. Signals

In the scope of the APIs, an event is the occurrence of something in MetaSolv Solution or in your application that is significant to the MetaSolv Solution user, such as:

- A request to export an LSR
- A request to send billing information
- A change in the status of the import of an LSC

A signal is a logical artifact that communicates information about an event.

Where appropriate or necessary, this documentation refers explicitly to gateway events or application events. If no such distinction is drawn, the information applies to either type of event. See "Understanding Events" for an explanation of the distinctions between application and gateway events.

Inbound Signals Vs. Outbound signals

The point of reference for this guide is the MetaSolv Solution product line. Therefore, when reading material about signals, the direction of the signal in relation to MetaSolv Solution determines whether it is an inbound or outbound signal. When MetaSolv Solution sends the signal, that signal is an outbound signal. When MetaSolv Solution receives the signal, that signal is an inbound signal. Where the documentation does not refer specifically to inbound or outbound signals, the information applies to both types of signals.

Synchronous Vs. Asynchronous

In the scope of this documentation, synchronous operations are those where the application that invokes the operation gets the results of the operation immediately upon the return of the call. No callback mechanism is used in this method. Asynchronous operations are those where control returns to the application that invokes the operation before the operation is acted upon, and the results (if any) are returned to the calling application after the operation is completed. The invoked application uses a callback mechanism to communicate the results to the invoking application. See "Synchronous and Asynchronous Invocation Modes" for more information about the synchronous and asynchronous modes of processing.

API Integration

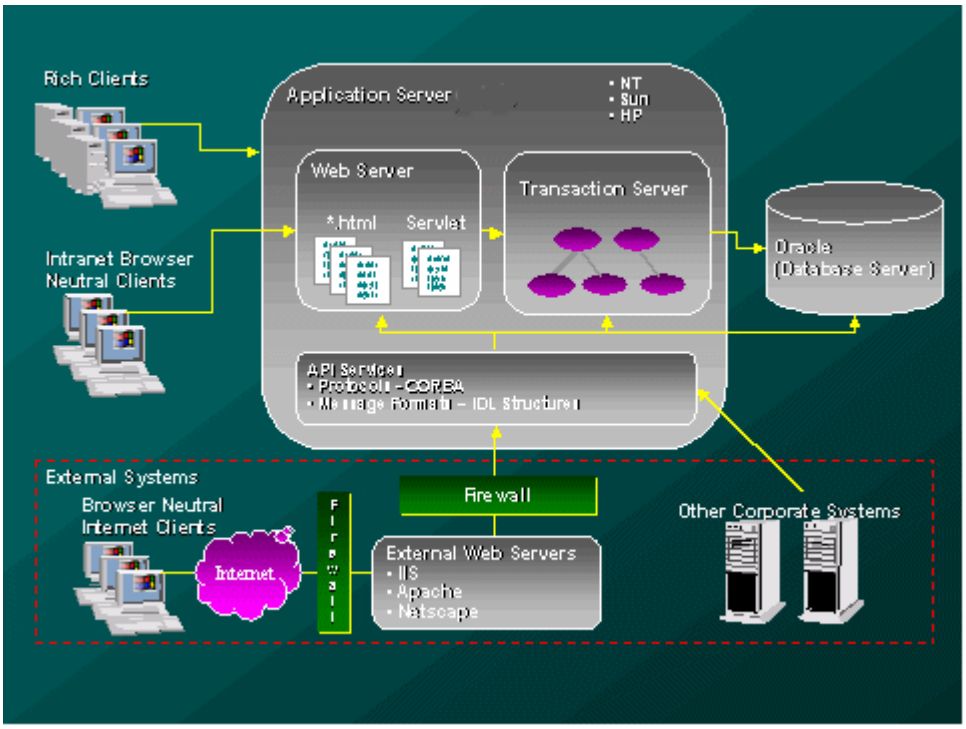
MetaSolv Solution provides APIs that allow the importing and exporting of data, through the use of the MetaSolv Solution Application Server, to the MetaSolv Solution database. This provides support for interconnection with third-party and legacy applications and allows development of customized interfaces to the MetaSolv Solution database. The APIs are built on the CORBA protocol. MetaSolv Solution defines the APIs using CORBA's IDL.

To provide interconnection, a MetaSolv Solution API provides a data pipe mechanism between your application and the MetaSolv Solution data model. This connection enables you to import and export data without programming language or methodology restrictions. Your software must implement an architecture that provides access to data and, if required, provides a mechanism for updating MetaSolv Solution data.

In addition to importing and exporting data, the APIs ensure the integrity of data in the MetaSolv Solution database by verifying that all imported data meets the MetaSolv Solution data rules. Using an API, MetaSolv Solution can be integrated into a customer's environment. Oracle consultants, third-party consultants, or customers themselves can complete the integration work.

Figure 1–1 shows the process by which the APIs communicate with client applications and other server applications.

Figure 1–1 API Communication Process Overview



MetaSolv Solution API Technical Overview

The MetaSolv Solution interface architecture provides APIs that enable access to specific information in the MetaSolv Solution database. This architecture meets requirements for customers connecting to MetaSolv Solution. Using this architecture, you or third-party developers can easily connect to MetaSolv Solution, providing add-on products and custom solutions.

MetaSolv Solution APIs are IDL files that provide a blueprint for communication between MetaSolv Solution and your software. The third-party server environment can be on any platform and operating system that supports CORBA.

The APIs are bidirectional, they send requests to other software and receive requests from other software. To initiate processing, MetaSolv Solution has defined an event mechanism that sends out pre-defined signals, called application events, and signals

defined by you or a third-party developer, called gateway events. See "Understanding Events" for more information.

Understanding Events

One of the most important tools provided by the APIs is the ability for your applications to integrate with the Work Management subsystem. This integration is provided by the exchange of events and signals between the APIs and your applications. An event is a significant occurrence within the workflow of either the Work Management subsystem or your application. A signal is the logical artifact used to communicate information about an event between MetaSolv Solution and your application.

Two types of events are implemented in the Work Management subsystem:

- Application events
- Gateway events

Application events are pre-defined within MetaSolv Solution and occur at fixed points in the workflow. Application events are always sent by MetaSolv Solution to external applications through an outbound signal that carries MetaSolv Solution-defined data pertaining to the event. The signal that represents an application event carries pre-defined data specific to that event.

Gateway events provide a powerful mechanism for you to insert hooks into the Work Management subsystem. The signal that represents a gateway event can carry only generic data such as a document reference for a document in the MetaSolv Solution database.

Except for the system-defined gateway events used by the PSR Ancillary API, all gateway events are defined by your application and set up in the MetaSolv Solution database using the user interface provided in the Work Management subsystem.

Note: See "The PSR Ancillary API and Smart Tasks" for more information about the system-defined gateway events supported by the PSR Ancillary API.

Unlike application events, which can only occur within MetaSolv Solution, gateway events can occur within either MetaSolv Solution or your application. Therefore, gateway events must be defined in the MetaSolv Solution database as either outbound or inbound events. Outbound gateway events occur within MetaSolv Solution and are communicated to your application through outbound signals. Inbound gateway events occur in your application and are communicated to the MetaSolv Solution Application Server through inbound signals.

Your application communicates the status of outbound gateway events to the MetaSolv Solution database through the APIs. These statuses indicate changes in the state of the event. The actual status values available for your use are defined in the event-signaling structure of the MetaSolv Solution IDL.

Gateway events must be tied to a task in a provisioning plan in the Work Management subsystem. When that provisioning plan is associated with a new order, the plan ensures that the Work Management subsystem sends or receives the gateway event at the point defined within the provisioning plan.

Table 1–2 summarizes the differences between application events and gateway events.

Table 1–2 Differences Between Application Events and Gateway Events

Difference	Application Events	Gateway Events
How Defined	Pre-defined in MetaSolv Solution	Defined by you (or a third-party developer) and added to the MetaSolv Solution database using the Work Management subsystem's user interface
Association	Tied to a specific MetaSolv Solution application event such as a button click or menu selection	Tied to a task in a provisioning plan in the Work Management subsystem
Signal Direction	Always outbound	Inbound or outbound as defined in the MetaSolv Solution database
Content	MetaSolv Solution-defined data specific to the event	Only generic data such as a document reference

For more information about Work Management integration, contact the Oracle representative at the implementation site.

Synchronous and Asynchronous Invocation Modes

The MetaSolv Solution interface architecture uses two invocation modes for operations:

- Synchronous
- Asynchronous

All external applications (those developed by you or a third party) that interact with the APIs are required to handle both invocation modes.

Synchronous Operations

In the scope of this documentation, synchronous operations are those where the application that invokes the operation gets the results of the operation immediately upon the return of the call.

The general rules for synchronous operations are as follows:

- All operations initiated by MetaSolv Solution or the API software against your application are synchronous. This means your application is required to return the results of the operation upon return of control.
- All operations your application invokes on an API server are synchronous, except for data import and export operations, which are asynchronous.

Asynchronous Operations

In the scope of this documentation, asynchronous operations are those where control returns to the application immediately and the results (if any) are returned to the invoking application at a later time. The APIs use a callback mechanism to implement this paradigm. The callback mechanism works as follows:

1. Your application creates a unique callback object, then passes that object to the appropriate API server along with the rest of the asynchronous operation's parameters. Your application then awaits return of control.
2. The API server implements the operation and immediately returns the call. Results of the operation are not returned at this time.

3. Control returns to your application, which now begins to listen to the callback object while it waits on the results.

> **Note:** If your application can handle multiple threads, the application can continue generating threads, if it remains available to accept and process the invocation of the callback object for each thread.

4. The asynchronous operation completes the requested task and returns the results to the API server.
5. The API server invokes an operation on the callback object to return the results to your application.
6. The callback object hands the results to your application

The general rule for asynchronous operations is that all operations involving movement of data to and from the MetaSolv Solution database, data import and export, are asynchronous.

> **Note:** See "Synchronous Operations" and "Asynchronous Operations" for more information about implementing synchronous and asynchronous operations.

Transaction Model Used By the APIs

The APIs use a transaction model; however, the APIs do not provide built-in support for nested or linked transactions. With two exceptions, the API servers provide an operation that external applications can invoke to generate a transaction object. The exception is the ICM API and the End User Billing API. These two servers provide all required transaction management functions internally.

Operations that involve movement of data into or out of the MetaSolv Solution database require the external application to supply a transaction object. The transaction object must support two operations:

- A commit operation that unconditionally applies all database changes that were performed during that transaction
- A rollback operation that cancels all database changes made during that transaction

The responsibility of organizing and managing units of work using the commit and rollback operations rests solely with external applications that use the APIs.

Transaction Objects

Each API server maintains an internal table of all the transaction objects it generates. The scope of a transaction object is ultimately limited to the lifetime of the API server process that created it and the lifetime of the MetaSolv Solution database instance. Transaction objects are re-usable but not portable. Therefore, the same transaction object may be used multiple times while performing operations on the API server that generated it, subject to the transaction lifespan limitations described earlier in this paragraph. You can only use a transaction object on the API server from which it was generated.

Determining the Role Your Application Performs

When developing an application to run against any of the APIs, it is very important to understand the roles that application will be performing. Applications can be developed against the APIs to perform in one of the following roles:

- Client only
- Server only
- Both client and server

Because of the significant differences between developing for these roles, it is important that you understand the differences between the roles. Before beginning development you should determine which role your application will perform in relation to the APIs.

For synchronous transactions each application's role remains constant throughout the transaction and is either the client role or the server role. See "Synchronous Operations" for more information.

The role each application plays is determined by the application that requests the service:

- The application that requests a service is the client
- The application that supplies the service is the server

For example:

- When your application invokes synchronous operations on the API servers to update the status of gateway events, your application is the client. In this case, your application requests the service and the API server supplies the service.
- When the Work Management subsystem sends an outbound gateway event to your application, your application is the server. MetaSolv Solution requests the service (in this case, WDISignal::eventOccurred) and your application supplies the service; in this case, whatever the application does when it receives that particular gateway event.

For asynchronous operations, the server role is also determined by which application requests the service, but the role each application plays can change, so role determination is not as simple as in synchronous operations. When invoking asynchronous API operations, perhaps your application will play any of the roles: client only, server only, or both. Therefore, external applications that invoke asynchronous operations against the APIs may have to be implemented with the capability to function as CORBA servers. See "Asynchronous Operations" for more information.

For example, when your application invokes an operation on the API server, your application is the client. However, when the API server invokes operations on a callback object that was provided by your application, the API server plays the client role and your application plays the server role.

Note: See "HelloAPI: Sample Application that Exports Data" for an example of an external application that plays both the client and server roles.

Importing and Exporting Using the APIs

The major processes supported by the APIs are the import and export of data.

By providing access to specific MetaSolv Solution data, APIs enable you to implement any required type of interface.

Two types of operations are provided by the APIs to enable external access to data stored in the MetaSolv Solution database.

- Data export operations are read-only and allow your application to extract data out of the database.
- Data import operations enable your applications to modify data stored inside the database.

The APIs provide operations that allow your application to obtain transaction handles to be used in data export and import operations.

Responsibilities When Developing With the APIs

An API provides a platform for integration, but it does not provide the complete functionality required. As a developer of external applications intended to work with the APIs, you should build in support for additional functionality as dictated by your application's unique requirements. The APIs expect your application to perform the following tasks:

- Transaction management: You must start and destroy the transaction object. In most cases, you must also define your own units of work and manage your application's interactions with the APIs through the *rollback* and *commit* operations provided by the API.
- Event handling: Design your applications to receive and process the application events and gateway events that the MetaSolv Solution clients send to your applications.

Also, if you are developing a gateway application, you typically need to build in support for service level agreement (SLA) functionality such as:

- Scheduling
- Field mapping/translation
- Defining protocols
- Transmission functionality retries, resends, recovery, and alternative transmission channels

Naming Conventions in the APIs

MetaSolv Solution provides three types of IDL files for each API. The IDL files define the interfaces your application can use to communicate with the MetaSolv Solution product line.

The types of IDL files provided with MetaSolv Solution are:

- The **WDI.IDL** file, a common API file distributed with each API. This file contains the highest level interface structures and operations used by all APIs.
- A **WDI***apiname***.IDL** file, where *apiname* represents the specific API name. This file contains the highest level application-specific interfaces and operations for the named API.
- One or more **WDIapiname**TYPES files, where *apiname* represents the specific API name. These files contain definitions of the data structures. There may be one WDIapinameTYPES.IDL file that contains common access information or any

number of additional **WDIapinameTYPES*n*.IDL** files, where *n* represents the types file number.

Note: The IDL types file for the Number Inventory API is named NITYPESE.IDL. IDL types files for the PSR Order Entry APIs are named PSRTYPES.IDL, PSRTYPES_v2.IDL, and PSRTYPES_v3.IDL.

IDL Versioning for MetaSolv Solution

MetaSolv Solution ensures the backward compatibility of the IDL. IDL is versioned as needed to support new functionality or to correct issues, leaving the original IDL backward compatible. However, in some cases we cannot provide this compatibility. Typically this occurs if an old function cannot be mapped to the new functionality or in cases where the original function signature was incomplete or unusable.

The syntax of the new versioned IDL files, API operations and structures includes a "_vX" at the end of the old name, where X represents a numeral; for example, importPSROrder is represented as importPSROrder_v2 in the versioned form. For new development use the most current version of an operation or structure. This means you would use operationname_v3 or structurename_v3 instead of operation or structurename_v2.

2

Developing Applications Using the APIs

This chapter introduces you to the steps involved in developing external applications that make use of the Oracle Communications MetaSolv Solution APIs. This chapter describes the various interaction paradigms and shows code examples for each.

This chapter also provides information on migrating from a previous release of MetaSolv Solution. See "Migrating to MetaSolv Solution 6.3.x from 5.x and 6.0.x" for more information.

This chapter assumes that you that you are familiar with the technical concepts presented in "Common Architecture".

Due to the variety of CORBA implementations available and the multitude of programming languages that a developer may use to do CORBA programming, it is not feasible to provide sample code for all possible combinations. The examples in this guide use the following environment:

- CORBA: JacORB
- Language: Java JEE

The basic principles and design patterns described in this guide remain valid regardless of the actual development environment used.

MetaSolv Solution 6.3.*x* uses JacORB 3.8, which includes CORBA versions 2.0, 2.3, 3.0, and 3.1. The CORBA ORB used for your applications that interface with the MetaSolv Solution API must support this standard. Typically that will mean you will need to upgrade your ORB to a later version and recompile your code. For most third party ORBs, this upgrade will not require code changes. Refer to your CORBA ORB vendors documentation to identify the steps required to support CORBA 2.4.

MetaSolv Solution Interface Architecture

The APIs have been designed to meet two primary goals:

- To enable external applications to perform on-demand data export and import operations on the database
- To allow external applications to tightly integrate with various modules and subsystems of MetaSolv Solution

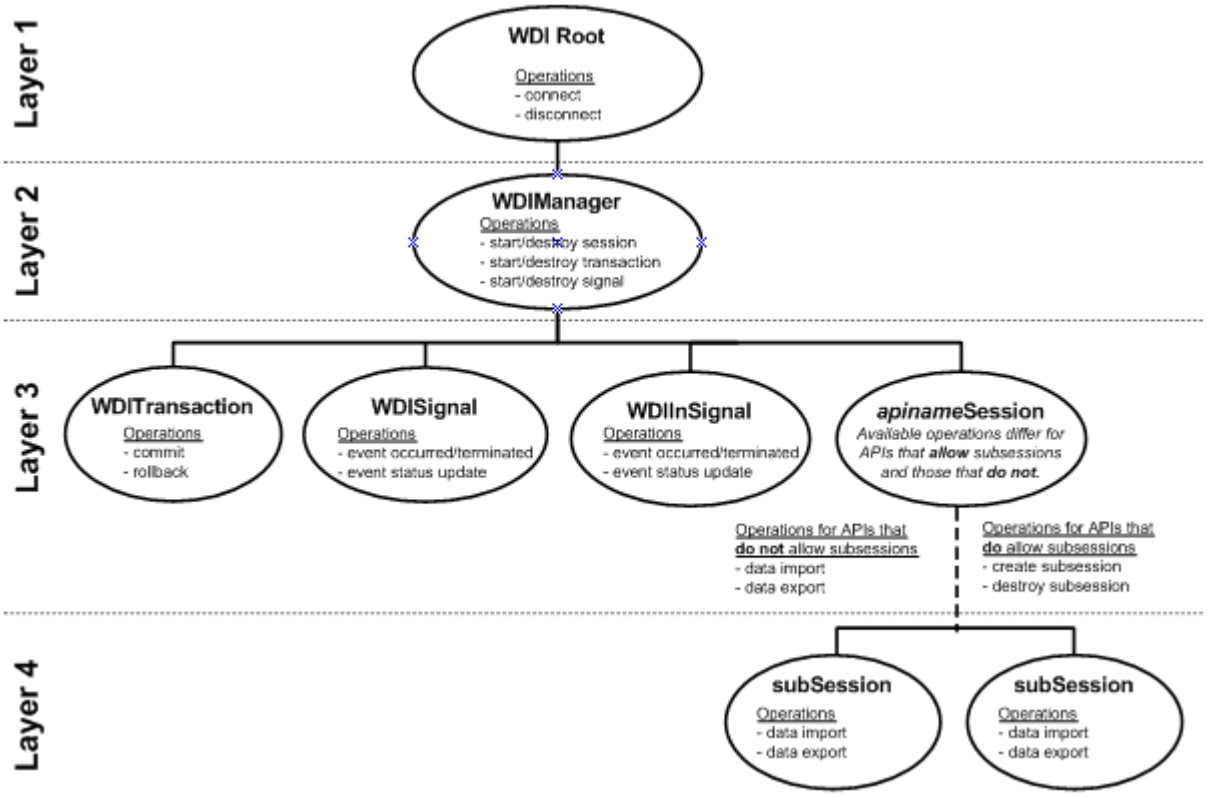
The MetaSolv Solution interface architecture is implemented on the Common Object Request Broker Architecture (CORBA) standard and is designed to be open and language-neutral. This enables you to develop your applications in any language that has CORBA bindings available and to deploy your applications on any platform that has CORBA support.

Design Architecture

Unlike traditional APIs, the MetaSolv Solution APIs are not packaged as object code or libraries. Instead, the API is delivered as a set of CORBA Interface Definition Language (IDL) files that are installed when the MetaSolv Solution Application Server is installed. CORBA implementation vendors supply IDL compilers that use these IDL files to generate language bindings for the desired implementation language (for example, Java or C++) and platform (for example, Windows NT or Oracle Solaris).

Figure 2–1 shows the API object architecture, which follows a layered, hierarchical approach. The interfaces at each layer support operations that yield object references to interfaces in the immediate subordinate layer (if any).

Figure 2–1 MetaSolv Solution API Design Architecture



The highest layer contains the Root interface, WDIRoot, which manages connections from client applications.

The second layer contains the Manager interface, WDIManager, that provides operations for session, transaction and signal management.

The third layer contains the various session interfaces. Typically, the operations that an external application is interested in are in the session interfaces. For example, operations that allow data export or import. However, some APIs are designed with a fourth layer. In these APIs, the session interface in the third layer provides operations that yield object references to interfaces in the fourth layer.

Deployment Architecture

The MetaSolv Solution Application Server may be hosted on the same machine where your application runs. However, in this case, the API class files must be loaded before any other class files to prevent errors from occurring in the operation of the APIs.

Relationship of APIs, API Server Names, and IDL Files

This object architecture is reflected in the structure of all the API IDL files.

Table 2–1 lists the APIs and their corresponding IDL files.

Table 2–1 Key IDL Files for MetaSolv Solution APIs

API	Server Name	Key IDL File
End User Billing API	Determined by you and entered in the MetaSolv Solution database through the Gateway Events window in the Work Management subsystem.	WDIPSRBIL.IDL
Inventory and Capacity Management API	DLRSERVER	WDIDLR.IDL
LSR API	LSRSERVER	WDILSR.IDL
PSR Ancillary API	PSRANCILLARYSERVER	WDIPSRANCILLARY.IDL
PSR Order Entry API	PSRSERVER	WDIPSR.IDL
Switch Provisioning Activation API	DLRSERVER	WDIDLR.IDL
Transport Provisioning Activation API	DLRSERVER	WDIDLR.IDL
Trouble Management API	TMSSERVER	WDITROUBLE.IDL
Work Management API	WMSERVER	WDIWM.IDL
Infrastructure API	INFRASTRUCTURESERVER	WDIINFRASTRUCTURE.IDL WDINETWORKLOCATION.IDL
Number Inventory API	NUMBERINVENTORYSERVER	WDINI.IDL

MetaSolv Solution APIs Require Instance References to Notification Objects

MetaSolv Solution APIs are written in Java and run under JDK 8. Due to JDK 8 requirements, the MetaSolv Solution APIs require your client-side application to pass an instance reference to the ORB, rather than a static reference. If your application passes a static reference to an ORB, the MetaSolv Solution APIs and the ORB cannot communicate.

Note: When a MetaSolv Solution API attempts to communicate through a static ORB reference, the error messages returned are often cryptic and unintuitive. For example, OrbixWeb raises a security violation rather than returning an exception indicating the requested object was not found.

Static ORB references are commonly passed because some IDL compilers' default behavior includes an implicit connect operation in generated constructors that is based

on a static ORB reference. However, the notification object must connect through an instance ORB reference or your client-side application and a server-side MetaSolv Solution API cannot both use the notification object.

Before compiling the IDL files for the MetaSolv Solution APIs, you should determine whether your IDL compiler's default behavior is to include an implicit connect operation in the generated constructors. If so, refer to your IDL compiler documentation to determine how to suppress inclusion of the connect operation in generated constructors.

Because the generated constructors intended for use with the MetaSolv Solution APIs must be compiled without a connect operation, your application must explicitly connect to each new notification object after instantiating the object. Make this connection using the *ORB.connect*(<notification instance reference>) operation.

Development Environment

In order to develop applications using the APIs you must have at a minimum these items installed on your workstation:

- A CORBA development environment that includes an IDL compiler and supporting classes and/or libraries, for example JacORB. The IDL compiler must be namespace aware because the MetaSolv Solution APIs use hierarchical naming conventions. Using an IDL compiler that is not namespace-aware causes naming collisions.
- A programming environment that includes a language compiler, runtime support classes and/or libraries and a debugger. This can be an integrated development environment (IDE) or could be composed of individual language components. For example, Java's JEE, the Sun JDK, and a C++ compiler.

When it is time to execute and test your applications, you must install the following:

- Your application and any software, resources, or services it requires
- The MetaSolv Solution client software with the API client components
- The MetaSolv Solution Application Server
- Oracle database server software and a MetaSolv Solution database that can be used for testing

Before Compiling IDL files

Before compiling the IDL files for a given API, you should place all the IDL files required for that API in the same folder, then execute your IDL compiler from within that folder.

Determining Which IDL Files Are Required for a Given API

1. Identify the key IDL file for the API. See Table 2–1 for more information.
2. Read the key IDL file and look for #include statements. These statements identify other IDL files that the IDL compiler must include when compiling.
3. Check each included IDL file for additional #include statements.

CORBA Development

This section describes the basic steps involved in developing CORBA applications. The intent of this section is to set the stage for the subsequent sections. The actual commands a user invokes to accomplish these steps varies depending on the development environment. Refer to your development environment's documentation for details.

The high-level steps involved in developing a CORBA application are as follows:

1. Compile the IDL.

   Locate the required API IDL files and run them through your IDL compiler. The compiler generates the following code in your language of choice:

   - Client-side stub: This is support code that you use to build a CORBA client application.
   - Server-side skeleton: This is support code that you use to build a CORBA server application.

2. Write code to implement the required IDL interface objects.
3. Write code to implement your client and server applications as applicable.
4. Compile your applications.
5. Register your CORBA server application with the ORB's implementation repository.

   Note: This step is essential in order for your server to start receiving CORBA calls from a client application.

6. Run your applications.

Implementation Patterns

This section introduces the basic patterns involved in developing external applications that utilize the MetaSolv Solution APIs. Real-world applications usually involve a combination of the basic patterns presented here.

Each implementation pattern is accompanied by a description that explains the purpose of the pattern, where it is used, and includes code fragments and information on how to write code to implement that pattern. The purpose of the code fragments is to illustrate the basic principles involved without having to address the specific requirements of individual APIs. Considerations for each API are provided in separate chapters later in this guide.

Each implementation pattern highlights a different concept; however, there is a considerable amount of overlap between the following sections because no one pattern can work in isolation from the rest.

Basic API Setup Pattern

This section describes the basic API setup pattern.

Purpose

This pattern illustrates the basic steps involved in API interactions. The successful completion of every operation invoked in this pattern is a precondition for working with the APIs.

When Used

When your application initiates the interaction with the MetaSolv Solution Application Server, as in a data export/import scenario, your application invokes the operations specified in the basic pattern, and the application server provides the implementation of these operations.

When MetaSolv Solution initiates the interaction with your application, such as when sending a gateway event to your application, MetaSolv Solution invokes the operations specified in this pattern. Your application provides the implementation of the interface and operations supporting the basic setup pattern.

For information about working with specific APIs, see the individual chapter later in this guide. For information about coordinating the MetaSolv Solution database and API security.

Description

The following Java-language code sample shows a sample implementation of the basic setup pattern. Table 2–2 lists the keys corresponding to the code samples.

```
package SampleCode.sample;
import java.io.*;
import org.omg.CORBA.*;
import MetaSolv.CORBA.WDI.WDIExcp;
import MetaSolv.CORBA.WDI.ConnectReq;
import MetaSolv.CORBA.WDI.WDITransaction;
import MetaSolv.CORBA.WDI.WDISignal;
import MetaSolv.CORBA.WDI.WDIInSignal;
import MetaSolv.CORBA.WDIDLR.*;
import MetaSolv.CORBA.WDIDLRTypes_v5.*;
/**
 * Hello API - Application Mainline
 * Description: Sample client application that demonstrates data
 *        export operation.
 *
 */
public class HelloAPI
{
  private static final String DLR_IOR_FILE_PROPERTY = "TmsDlrIorFile";
  private static ORB orb;
  public static ORB getORB() {
    return orb;
  }
  public static void main(String [] args)
  {
    System.out.println(System.getProperties().toString());
(1)    orb = ORB.init(args, null);
    Utils.initORB(orb);
    try {
      int circuitId = 21;  // (pk of design_layout_report)
      int issueNo = 1;
      DLR aDLR = getCircuitIssue(circuitId, issueNo);
      if (aDLR != null)
```

```
        System.out.println("Circuit ECCKT:" + aDLR.dlrAdminInfo.ECCKT);
    } catch (Throwable t) {
      System.out.println("Error: "+ t.toString());
    }
    System.out.println("Exiting application");
    System.exit(0); // Some ORBs start non-daemon threads running.
  }
  public static DLR getCircuitIssue(int circuitId, int issueNo) throws Exception
  {
    DLR aDLR = null;
    // Connect to the DLR API Server and construct a proxy for root object
    String iorfile = System.getProperties().getProperty(DLR_IOR_FILE_PROPERTY);
// Set a system property on command line using -D (for Sun) or /d: (for MS)
    if (iorfile == null)
       throw new Exception("'" + DLR_IOR_FILE_PROPERTY +
"' system property not set on command line.");
(2)    System.out.println("IOR file="+iorfile);
    String ior = Utils.readIOR(iorfile);
    System.out.println("DLR IOR="+ior);
    org.omg.CORBA.Object obj = orb.string_to_object(ior);
    WDIRoot aWDIRoot = (WDIRoot)WDIRootHelper.narrow(obj);
(3)    MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
    req.userName = "ASAP";
    req.passWord = "ASAP";
    System.out.println("Connecting to MetaSolv Solution API Server...");
(4)    WDIManager aWDIManager = aWDIRoot.connect(req);
    try {
      System.out.println("Starting transaction...");
(5)      WDITransaction aWDITransaction = aWDIManager.startTransaction();
      try {
        System.out.println("Starting session...");
(6)        DLRSession aDLRSession = aWDIManager.startDLRSession();
        try {
(7)          WDIExampleNotificationImpl aWDINotificationImpl =
new WDIExampleNotificationImpl();
          // Need to connect the NotificationImpl Object to the ORB for callbacks.
          MetaSolv.CORBA.WDIDLR.WDINotification aWDINotification =
               (MetaSolv.CORBA.WDIDLR.WDINotification)
MetaSolv.CORBA.WDIDLR.WDINotificationHelper.narrow(Utils.connect(aWDINotificationI
mpl));
(8)          System.out.println("Sending request...");
          aDLRSession.getDLR_v5(aWDITransaction, aWDINotification,
new DLRRequest(circuitId, issueNo));
          System.out.println("Sent request. Waiting on notify callback ...");
          aWDINotificationImpl.waitForResponse();
          aDLR = aWDINotificationImpl.getDLR_v5();
(9)          // Do not need to commit for exporting data but to be consistent
          try {
            aWDITransaction.commit();
          } catch (Throwable t) {
            System.out.println("Error: " + t.toString());
          }
          if (aWDINotificationImpl.hasErrors())
            aWDINotificationImpl.printErrors();
(10)          // Need to disconnect to prevent memory leaks
          Utils.disconnect(aWDINotification);
        }
(11)        catch (Throwable t)
        {
```

```
          System.out.println("Error " + t.toString());
        } finally {
          aWDIManager.destroyDLRSession(aDLRSession);
        }
(12)      } catch(Throwable t) {
        System.out.println("Error " + t.toString());
      } finally {
        aWDIManager.destroyTransaction(aWDITransaction);
      }
    }
(13)    finally {
      aWDIRoot.disconnect(aWDIManager);
    }
    return aDLR;
  }
}
```

Table 2–2 Keys for Basic API Setup Pattern Example Code

Key	Description
(1)	Initialize the Object Request Broker (ORB). The actual call used to do so varies depending on the ORB vendor.
(2)	Establish a CORBA connection with your server application. The sample code connects to DLRSERVER that is running on a machine with TCP/IP host name MetaSolv Solutionapihost. The bind operation is specific to the CORBA implementation.Information about the ORB connections can be found in the *MSS System Administrator's Guide*.
(3)	Create a connection object to use when obtaining an instance of the WDIManager interface from DLRSERVER. This instance is actually a proxy object that is created in the application's program space, and not the actual instance of the interface that is created on the server. The proxy object simulates the real instance, and forwards all operation invocations to the real instance using the CORBA protocol. See *MSS Security Guide* for security information. It is mandatory that you populate the user name and password fields on the ConnectReq object when you create it and pass it to the WDIRoot object.
(4)	Perform the connect operation to obtain an instance of the WDIManager interface. This instance provides specialized operations provided by the MetaSolv Solution Application Server; in this example, the DLR server.
(5)	Obtain a transaction handle from DLRSERVER. This step is only required when the invoked operation requires a transaction handle, as defined in the interface specification. For example, WDIDLR.IDL requires transaction handles, and WDIPSRBIL.IDL does not.
(6)	Obtain an instance of the DLRSession interface. In most of the APIs, the session interface supports all data export and import operations.
(7)	Instantiate a WDINotification object and tie it to the ORB so the calling server can make notification callbacks.
(8)	The core activity, the actual work, performed by your application is performed at this point. All of the actual work to be performed must be done within this section of the code.
(9)	Call the commit operation so the API commits your changes to the database.
(10)	Destroy the instance of the session interface. This begins the process of tearing down the communications infrastructure built in above Steps (2)–(6). Tearing down the infrastructure must be done in the reverse order in which it was built.
(11)	Clean up the objects no longer used.

Table 2–2 (Cont.) Keys for Basic API Setup Pattern Example Code

Key	Description
(12)	Destroy the transaction handle. **Warning:** You must always perform all transaction management steps, such as commit or rollback, prior to destroying the transaction handle.
(13)	Destroy the instance of WDIManager interface. **Warning:** Establishing a connection with a CORBA server [performed above in Step (2)] is an expensive operation in terms of performance. To keep this overhead to a minimum, consider performing server connection operations only at application startup time. However, your application's design should be driven by your specific requirements.

Synchronous Interaction Pattern

This section describes the synchronous interaction pattern.

Purpose

This pattern illustrates synchronous interaction between your application and MetaSolv Solution. For a complete explanation of synchronous interaction, see "Synchronous Operations" for more information.

When Used

All operations initiated by MetaSolv Solution against your application are synchronous. Except for data import and export operations, which are asynchronous.

Description

See "Basic API Setup Pattern" for examples of synchronous operation invocations. For instance, the following code fragment shows an application invoking a synchronous operation startTransaction.

```
// Obtain a transaction handle from MetaSolv Solution API server.
try {
    WDITransaction aWDITransaction = aWDIManager.startTransaction();
}
catch (WDIExcp ex) {
    System.err.println("Error getting transaction handle: " + ex.getMessage());
}
```

The result of the operation invocation, in this case, a transaction handle, is immediately available to the caller upon completion of the operation. If the invocation fails, the API immediately raises an exception and the result, the exception that indicates failure, is immediately available to the caller upon return of control. See "Error Handling Pattern" for more information.

In the scenario where the MetaSolv Solution client communicates a gateway event to your application, the MetaSolv Solution client follows the steps in "Basic API Setup Pattern" with the exception of Step (5) of that pattern. In this scenario, your application implements all the required operations in synchronous mode. Upon invoking a synchronous operation on your software, the MetaSolv Solution client displays the hourglass icon to the user and suspends user interaction until the call returns.

Warning: If your application fails to return the call to the MetaSolv Solution client as expected, that failure causes MetaSolv Solution to hang. The user must reboot.

See "HelloGateway: Sample Application that Handles Application and Gateway Events" for a complete working example of the synchronous interaction pattern.

Asynchronous Interaction Pattern

This section describes the asynchronous interaction pattern.

Purpose

This pattern illustrates the asynchronous mode of interaction between your application and MetaSolv Solution. The MetaSolv Solution Application Server mandates the use of this pattern by your application when invoking the import and export operations.

When Used

The asynchronous mode of interaction is only used when your application interacts with the APIs to request the export or import of data. All data export and import operations involving the APIs are defined as asynchronous.

To determine from the IDL whether an operation is asynchronous, look at the operation specification in the corresponding IDL file. If that specification defines one of the operation parameters as type WDINotification, then the operation is asynchronous. For such operations, the return type is void, that is, they do not return anything to the caller.

Description

Asynchronous interaction is achieved through a callback mechanism. The caller of the operation, your application, creates a unique callback object and passes it to the provider, the MetaSolv Solution Application Server, along with the rest of the operation's parameters. To return the result of the operation, the provider invokes operations on that callback object.

The crux of implementing code to handle asynchronous mode interactions is to develop a robust mechanism to handle callback invocations from the API. The basic requirement here is that you must provide an implementation of the WDINotification interface as defined by the particular API's IDL file. When an asynchronous operation on that API is invoked, the API calls one of the operations defined in the WDINotification interface.

For example, if your application invokes the getDLR_v5 asynchronous operation on the DLRSERVER, you must provide an implementation of the WDINotification interface as defined in the WDIDLR.IDL file. A fragment of this IDL file is reproduced below. In that fragment, notice that the only operations that the DLRSERVER would invoke on your application's callback object are: DLRGetSucceeded_v5 and DLRGetFailed_v5.

Table 2–3 lists the keys for WDINotification Example IDL.

Table 2–3 Notes for WDINotification Example IDL

Key	Description
(1)	Callback upon successful completion of the operation
(2)	Callback to indicate failure of the operation

Example 2–1 WDIDLR.IDL fragment

```
// CCM#40525   Circuit Query API
void        getDLRsByQuerySucceeded_v3(in MetaSolv::CORBA::WDIDLRQueryTypes_
v3::DLRQuery aDLRQuery,
 in MetaSolv::CORBA::WDIDLRQueryTypes_v3::DLRResultSeq aDLRResults);
void        getDLRsByQueryFailed_v3(in MetaSolv::CORBA::WDIDLRQueryTypes_
v3::DLRQuery aDLRQuery,
 in WDI::WDIErrSeq aWDIErrSeq);
 /// Deprecated - DLRGetSucceeded_v2, DLRGetFailed_v2 in future release will be
removed.
 /// You should use the latest version of this method.
void        DLRGetSucceeded_v2(in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest
aDLRRequest,
 in MetaSolv::CORBA::WDIDLRTypes_v2::DLR aDLR);
void        DLRGetFailed_v2(in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest
aDLRRequest,
 in WDI::WDIErrSeq aWDIErrSeq);
// CCM#40030
 void        DLRGetSucceeded_v3(in MetaSolv::CORBA::WDIDLRTypes_v3::DLRRequest
aDLRRequest,
                    in MetaSolv::CORBA::WDIDLRTypes_v3::DLR aDLR);
void        DLRGetFailed_v3(in MetaSolv::CORBA::WDIDLRTypes_v3::DLRRequest
aDLRRequest,
 in WDI::WDIErrSeq aWDIErrSeq);
// CCM#30450
void        DLRGetSucceeded_v4(in MetaSolv::CORBA::WDIDLRTypes_v4::DLRRequest
aDLRRequest,
 in MetaSolv::CORBA::WDIDLRTypes_v4::DLR aDLR);
void        DLRGetFailed_v4(in MetaSolv::CORBA::WDIDLRTypes_v4::DLRRequest
aDLRRequest,
 in WDI::WDIErrSeq aWDIErrSeq);
// CCM#43899
(1) void        DLRGetSucceeded_v5(in MetaSolv::CORBA::WDIDLRTypes_v5::DLRRequest
aDLRRequest,
                    in MetaSolv::CORBA::WDIDLRTypes_v5::DLR aDLR);
(2) void        DLRGetFailed_v5(in MetaSolv::CORBA::WDIDLRTypes_v5::DLRRequest
aDLRRequest,
 in WDI::WDIErrSeq aWDIErrSeq);

void        switchGetSucceeded_v2(in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest
aDLRRequest,
 in MetaSolv::CORBA::WDIDLRTypes_v2::DLRSwitchTranslation aDLRSwitchTranslation);
void        switchGetFailed_v2(in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest
aDLRRequest,
 in WDI::WDIErrSeq aWDIErrSeq);
void
 in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest aDLRRequest,
 in MetaSolv::CORBA::WDIDLRTypes_v2::DLREndUserSpecialTrunkTranslation
aDLREndUserSpecialTrunkTranslation);
void
 in MetaSolv::CORBA::WDIDLRTypes_v2::DLRRequest aDLRRequest,
```

```
 in WDI::WDIErrSeq aWDIErrSeq);
```

Each asynchronous operation defined in an API's session interface has two counterparts in the WDINotification interface: one to callback upon successful completion of the operation and the other to indicate failure. In short, although the basic rules of interaction are consistent across the MetaSolv Solution APIs, each specific API defines this interface differently.

The following Java-language code fragment illustrates how the callback mechanism can be implemented.

Table 2–4 lists the keys for the callback mechanism implementation example codes.

```
(1)public class WDIExampleNotificationImpl extends WDINotificationPOA
{
  DLR aDLR = null;
  WDIError[] aWDIErrSeq = null;
  boolean done = false;
  public WDIExampleNotificationImpl ()
  {
    super();
  }
   /**
  * Gets the DLR object that was returned from the API
  * @return DLR
  */
(2)  public DLR getDLR()
  {
    return aDLR;
  }
   /**
  * Checks to see if any errors were returned by the API
  * @return boolean true if there are errors, false otherwise
  */
  public boolean hasErrors()
  {
    if (aWDIErrSeq == null)
      return false;
    return true;
  }
   /**
  * Prints any errors to the console
  */
  public void printErrors()
  {
    // Should use hasErrors to check if there are any errors
    for(int i = 0; i < aWDIErrSeq.length; i++)
    {
      System.out.println("Code: " + aWDIErrSeq[i].code
                  + "  Reason: " + aWDIErrSeq[i].reason);
    }
  }
(3)  // Utility method: Force thread to wait on callback from DLRSERVER
  public synchronized void waitForResponse()
  {
    try { if (!done) wait(); } catch (InterruptedException e){}
  }
(4)  // Interface methods: Implement all the operations defined in the
  // WDINotification interface as defined in WDIDLR.IDL file.
  // Note: Provide trivial implementations for methods that the server will not
```

```
  // invoke in this scenario.
  public int getMaximumReturnedRows()
  {
    return 0;
  }
  public void getDLRsByServiceRequestSucceeded_v2(int documentNumber,
          MetaSolv.CORBA.WDIDLRQueryTypes_v2.DLRResult[] results)
  {
  }
  public void getDLRsByServiceRequestFailed(int documentNumber,
          MetaSolv.CORBA.WDI.WDIError[] aWDIErrSeq)
  {
  }
  public void getDLRsByCircuitSucceeded_v2(int circuitId,
          MetaSolv.CORBA.WDIDLRQueryTypes_v2.DLRResult[] results)
  {
  }
  public void getDLRsByCircuitFailed(int circuitId, MetaSolv.CORBA.WDI.WDIError[]
          aWDIErrSeq)
  {
  }
  public void getDLRsByQuerySucceeded_v3(MetaSolv.CORBA.WDIDLRQueryTypes_
v3.DLRQuery
          aDLRQuery, MetaSolv.CORBA.WDIDLRQueryTypes_v3.DLRResult[] results)
  {
  }
  public void getDLRsByQueryFailed_v3(MetaSolv.CORBA.WDIDLRQueryTypes_v3.DLRQuery
aDLRQuery,
          MetaSolv.CORBA.WDI.WDIError[] aWDIErrSeq)
  {
  }
  // DLRSERVER callback for the case when getDLR operation is successful
(5)  public synchronized void DLRGetSucceeded_v5(MetaSolv.CORBA.WDIDLRTypes_
v5.DLRRequest
   aDLRRequest, MetaSolv.CORBA.WDIDLRTypes_v5.DLR aDLR)
  {
    System.out.println("DLRGetSucceeded notification called");
    this.aDLR = aDLR;
    this.aWDIErrSeq = null;
    done = true;
    try { notifyAll(); } catch (Throwable t){}
  }
(6)  // DLRSERVER callback for the case when getDLR operation fails
  public synchronized void DLRGetFailed_v5(MetaSolv.CORBA.WDIDLRTypes_
v5.DLRRequest aDLRRequest,
          MetaSolv.CORBA.WDI.WDIError[] aWDIErrSeq)
  {
    System.out.println("DLRGetFailed notification called");
    this.aDLR = null;
    this.aWDIErrSeq = aWDIErrSeq;
    done = true;
    try { notifyAll(); } catch (Throwable t){}
  }
```

Table 2–4 Notes for Callback Mechanism Implementation Example Code

Key	Description
(1)	Support for server callback is implemented by extending (sub-classing) the WDINotificationPOA class. The IDL compiler generates this class when it compiles the WDIDLR.IDL file to create Java language bindings. The specialized class (sub-class) is named WDIExampleNotificationImpl. This class implements all the operations specified in the DLRSession interface defined in WDIDLR.IDL.
(2)	Since the class in this example is the one that receives the callback and the results of the getDLR_v5 operation from DLRSERVER, an **accessor** operation is provided to enable the users of the class to retrieve the results.
(3)	This method may be called by users of the class to suspend activity in their threads pending callback invocation from the API. Towards that end, this sample code uses method synchronization mechanisms supported by the Java language. Alternate synchronization mechanisms can be used to achieve the same effect.
(4)	This sample provides trivial implementations for operations that are defined in the DLRSession interface, but will never be invoked by the DLRSERVER for the export operation that this sample invokes. This is necessary because the Java compiler requires these implementations since the compiler defines the base class for this sample, WDINotificationPOA, as an abstract class.
(5)	The code sample implements the DLRGetSucceeded_v5 operation that the DLRSERVER invokes if the getDLR_v5 operation invocation is successful. The server returns results in the second parameter of this operation through an object of type WDIDLRTYPES_v5.DLR. In the code sample, the results are stored in a private attribute.
(6)	The code sample implements the DLRGetFailed_v5 operation invoked by the DLRSERVER if the getDLR operation invocation resulted in failure. The reasons for the failure are communicated to the class through the second parameter of this operation, an object of type WDI.WDIError.

This example may easily be extended to provide support for other asynchronous DLRSERVER operations by building in non-trivial implementations of the other callback operations.

The following code fragment shows how an application might invoke an asynchronous operation on an API. This fragment continues from the previous example. At this point in the code, your application invokes the getDLR_v5 operation on DLRSERVER. However, the code must first obtain a transaction handle from DLRSERVER since the getDLR_v5 operation requires that you supply one. Next, the code creates a proxy for the DLRSession interface. Table 2–5 lists the keys for the for extended callback mechanism implementation example codes.

```
try {
    WDITransaction aWDITransaction = aWDIManager.startTransaction();
        try {
            DLRSession aDLRSession = aWDIManager.startDLRSession();
            try {
(1)                 WDIExampleNotificationImpl
                     aWDINotification = new WDIExampleNotificationImpl();
(2)                 aDLRSession.getDLR_v5 (aWDITransaction, aWDINotification,
                     new DLRRequest(circuitId, issueNo));
                System.out.println("Sent request to " + servername +
(3)                     ". Waiting on notify callback ...");
                try {
                    aWDINotification.waitForResponse();
                    aDLR = aWDINotification.getDLR_v5 ();
                }
```

```
catch(Exception ex) {
    System.err.println("getDLR()_v5 failed: " + ex);
}
```

Table 2–5 Notes for Extended Callback Mechanism Implementation Example Code

Key	Description
(1)	Create a new instance of the callback object that was implemented previously.
(2)	Invoke the getDLR_v5 operation, supplying the callback object along with the other input parameters to the operation. The operation returns without the results, as it should.
(3)	Next, suspend activity to wait on response from DLRSERVER in the form of a callback operation invocation. **Warning:** If the MetaSolv Solution Application Server terminates for any reason while a callback is pending, the pending call back will never be satisfied unless your application takes steps to clean up and retry the operation (if required) when the server is restarted.

Your application that performs data import/export operations using an API could fire off a number of requests in a burst, without waiting on the results. It would subsequently receive a number of results through callback invocations. Your application would then perform the required steps to collate the results.

The third-party application's developer must also compile the application code with the server-side skeleton and implementation code.

CORBA Client/Server Pattern

This section describes the CORBA client/server pattern.

Purpose

The intent of this section is to show, at a high level, what it takes to develop a CORBA client and CORBA server and to highlight the differences between these two.

> **Note:** You should review the documentation provided by your ORB vendor for an in-depth discussion on this topic as it relates to your ORB environment.

The MetaSolv Solution interface architecture is built on the CORBA standard. As the developer of an application developed to use the APIs, you must determine whether your application will be required to play the roles of a CORBA client, a CORBA server, or both.

To define these roles in simple terms, the application that invokes an operation on a CORBA interface is the client. The application that implements the invoked operation is the CORBA server. See "Determining the Role Your Application Performs" for more information.

When MetaSolv Solution initiates the interaction with your application, your application plays the role of the server.

> **Note:** The client code and the server code do not have to run in separate program spaces. The same application can play the role of a client and server.

When Used

Your application plays the role of a client when it invokes operations on the MetaSolv Solution Application Server, as in the following scenarios:

- When invoking data export/import operations
- When updating status of gateway events received from MetaSolv Solution clients
- When invoking operations to communicate inbound signals

Your application plays the role of a server in these scenarios:

- When handling outbound signals, whether the signals represent application events or gateway events
- When handling callback operations from the MetaSolv Solution Application Server

Your application plays both roles when it performs functionality for both of the scenarios described above. In practice, such dual-mode applications are more common than pure client or server applications. Common examples are:

- An application that invokes an asynchronous operation on an API
- An application that receives a gateway event from a MetaSolv Solution client and subsequently updates the status of that event based on work performed outside MetaSolv Solution.

> **Note:** Using a CORBA naming service implementation avoids the necessity of hard-coding the physical location of MetaSolv Solution into external applications. The naming service provides white pages functionality allowing clients to query, at runtime, the location of servers they wish to use.

Signal Handling Pattern

This section describes the signal handling pattern.

Purpose

Events are the means by which external applications can integrate with the Work Management subsystem. Signals are the mechanisms used to communicate events between MetaSolv Solution and the external applications. The signal-handling pattern describes how external applications can implement signal handling.

> **Note:** The Trouble Management API uses the fundamental concepts of the signal handling pattern that are implemented by the other APIs. However, the Trouble API requires a different set of attribute values to uniquely identify an instance of an event within a trouble ticket. Using this variation of the signaling mechanism enables the Trouble Management API to support multiple concurrent events for a given trouble ticket.

When Used

MetaSolv Solution clients use outbound signals to communicate application events and gateway events to external applications. External applications use inbound signals to communicate external events to MetaSolv Solution.

Description

A general observation to be made here is that all operations involved in handling signals are synchronous. Because of the inherent difference between inbound and outbound signals and between application and gateway events, the following section is divided as indicated below:

- Outbound Signals – Gateway Events
- Outbound Signals – Application Events
- Inbound Signals

The types of signaling supported by the various APIs can vary. Some APIs support one or more kinds while there are some for which signaling is not applicable.

General Remarks On Outbound Signals

Whenever your application handles outbound signals, it plays the role of a CORBA server. In such a scenario, you should design your application to handle multiple incoming requests. Remember that there may be any number of client machines in your environment that could potentially communicate events to your application.

A MetaSolv Solution client only establishes connection with your application once, when the first outbound signal is sent to your application. If your application were to terminate subsequently, the client would get an error the next time it sends an outbound signal. This requires a restart of the client. In other words, your application should be running during all of the business hours when you expect clients to be active.

Outbound Signals – Gateway Events

These signals originate from MetaSolv Solution clients and carry a standard data payload that is bound for your application. The structure of this payload is defined in the WDIEvent data structure in file WDI.IDL, which is reproduced below.

```
struct WDIEvent
{
    long   eventVersion;
    string eventName;
    long   documentNumber;
    long   taskNumber;
    long   servItemID;
    string userID;
};
```

The **eventName** field identifies the event. The value populated in this field is picked up from the gateway event definition in the MetaSolv Solution database. The **documentNumber** field is a database-generated sequence number that uniquely identifies a service request in the database.

The example code shown in "Example Code for Implementing WDIManager and WDISignal Interfaces" demonstrates how an application can handle outbound signals. The goal of the sample code is to develop a CORBA server that handles all the operations that a client may invoke on it in order to communicate gateway events. The first detail to be worked out is what specific operations should be implemented out of the complete IDL file. Table 2–6 lists the mandatory operations for all MetaSolv Solution APIs. The Trouble Management API also requires implementation of the startSignal2 and destroySignal2 operations.

Table 2–6 Outbound Gateway Event Operations Required For All APIs

Interface	Operations	Remarks
WDIRoot	connect	A client requests to establish a connection.
WDIRoot	disconnect	MetaSolv Solution requests to destroy the connection.
WDIManager	startSignal	MetaSolv Solution indicates start of signal. Your application generates a WDIEvent instance.
WDIManager	destroySignal	MetaSolv Solution indicates end of signal.
WDISignal	eventOccurred	MetaSolv Solution indicates the occurrence of an event within the Work Management subsystem and passes the data payload.
WDISignal	eventTerminated	MetaSolv Solution indicates the user opted to bypass processing for this event. This indicates that your application is to stop processing this event.

Note: Your compiler may force you to supply placeholder implementations for the remaining operations defined in the IDL files for these interfaces

The actual API used in the implementation would depend on how the gateway event is defined in MetaSolv Solution.

The other half of handling gateway events is the work of updating status of events in database. It is your application's responsibility to update event statuses, based on external activity that is applicable to the situation.

The following list identifies the main steps in developing applications that update event statuses:

1. Write server mainline. See "CORBA Client/Server Pattern" for the sample server mainline code.
2. Implement WDIRoot. See "CORBA Client/Server Pattern" for the sample code.
3. Implement WDIManager and WDISignal interfaces. See "Example Code for Implementing WDIManager and WDISignal Interfaces" for the sample code.
4. Implement code to update event status. See "Example Code for Updating Status of Events" for the sample code.

The following Java-language code fragment shows a sample implementation of the WDIManager and WDISignal interfaces. Table 2–7 lists the keys for the WDIManager and WDISignal Implementation example codes.

Example Code for Implementing WDIManager and WDISignal Interfaces

```
(1)package SampleCode.sample;
import MetaSolv.CORBA.WDI.WDIExcp;
import MetaSolv.CORBA.WDI.ConnectReq;
import MetaSolv.CORBA.WDI.WDITransaction;
import MetaSolv.CORBA.WDI.WDISignal;
import MetaSolv.CORBA.WDI.WDIInSignal;
import MetaSolv.CORBA.WDI.WDIEvent;
import MetaSolv.CORBA.WDI.WDIStatus;
import MetaSolv.CORBA.WDI.WDIError;
/**
```

```
 * Hello Gateway -- Gateway Event Handler
 * Description: This class implements the WDISignal interface as defined in the
WDI.IDL
 *       When a gateway event occurs, MetaSolv Solution Client invokes the
following operations on
 *       this interface: eventOccurred, eventTerminated.
 * @version 5.0.0
 */
public class WDIGatewaySignalImpl extends MetaSolv.CORBA.WDI.WDISignalPOA
{
  public WDIGatewaySignalImpl() {
     super();
  }
  // This method is invoked when a MetaSolv Solution client transmits a gateway
  // event to our server.
(2)  public void eventOccurred(WDIEvent aWDIEvent)
  {
    System.out.println("WDIGatewaySignalImpl.eventOccurred");
    // Start a new thread to handle this gateway event.
    new RequestThread(aWDIEvent).start();
    // In practice, this method should not return until the
    // event is successfully placed in persistent storage.
  }
  // This method is invoked when a MetaSolv Solution client requests that a
  // previously transmitted event be cancelled.
(3)  public void eventTerminated(WDIEvent aWDIEvent)
  {
    System.out.println("WDIGatewaySignalImpl.eventTerminated");
    // In practice, this method should not return until the request
    // is safely persisted in a queue.
  }
  // NOTE: The following three operations on WDISignal interface are
  //   implemented by the MetaSolv Solution API servers. However, we need to
  //   provide trivial implementations to satisfy the compiler.
(4)  public WDIStatus eventInProgress(WDITransaction aWDITransaction, WDIEvent
aWDIEvent)
  {
    return null;
  }
  public WDIStatus eventCompleted(WDITransaction aWDITransaction, WDIEvent
aWDIEvent)
  {
    return null;
  }
  public WDIStatus eventErrored(WDITransaction aWDITransaction,
              WDIEvent aWDIEvent, WDIError[] aWDIErrorSeq)
  {
    return null;
  }
  /**
   * @since 5.0.0
   */
  public WDIStatus eventInProgress2(WDIEvent aWDIEvent)
  {
    return null;
  }
  /**
   * @since 5.0.0
   */
  public WDIStatus eventCompleted2(WDIEvent aWDIEvent)
```

```
  {
    return null;
  }
  /**
   * @since 5.0.0
   */
  public WDIStatus eventErrored2(WDIEvent aWDIEvent, WDIError aWDIError[])
  {
    return null;
  }
}
```

Table 2–7 Keys for WDIManager and WDISignal Implementation Example Codes

Key	Description
(1)	Extend the WDIManager interface. Subclass the WDISignalPOA class generated by the IDL compiler to create the specialized class that provides implementations of the required operations on WDIManager interface.
(2)	Implement WDISignal interface. Subclass the WDISignalPOA class generated by the IDL compiler to create the specialized class that provides implementations of the required operations on WDISignal interface. **Note:** The sample code provides placeholder implementations for operations that are not invoked in this scenario.
(3)	Implement *eventOccurred* operation. This is where you would want to store this event in some form of persistent storage for future processing. It is recommended that you return from this call only upon successful completion of the persistence operation. The sample code starts a new thread instance to process this event.
(4)	Implement *eventTerminated* operation. This operation is invoked to indicate that the user has chosen to bypass this gateway event. In practice, you would remove the specified event from persistent storage and desist from further processing of that event. **Note:** The sample code provides placeholder implementations for operations that are not invoked in this scenario.

The following sample code addresses the second part of this implementation, updating the status of events.

Example Code for Updating Status of Events

```
String hostname = "MetaSolv Solutionapihost";  // machine name of MetaSolv
Solution API host
String servername = "DLRSERVER";  // MetaSolv Solution API CORBA server name
try {
(1)    WDIRoot aWDIRoot = WDIRootHelper.bind(":"+servername, hostname);
}
catch (SystemException se) {
    System.out.println("Unable to bind to server: " + se);
}
ConnectReq req = new ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = null;
try {
    aWDIManager = aWDIRoot.connect(req);
}
catch (WDIExcp ex) {
    System.err.println("connect failed: " + ex.reason);
```

```
}
try {
    WDITransaction aWDITransaction = aWDIManager.startTransaction();
    try {
        WDISignal aWDISignal = aWDIManager.startSignal();
        try {
(2)             WDIStatus myStatus = aWDISignal.eventInProgress(aWDITransaction,
                 aWDIEvent);
```

Table 2–8 Keys for WDIManager and WDISignal Implementation Example Codes

Key	Description
(1)	Connect to DLRSERVER, since the gateway event was defined to use the Inventory and Capacity Management (ICM) API interface.
(2)	Set event status to In Progress to indicate that the external activity triggered by this event is in progress. Following this, you can set the event status to Completed or Errored using operations defined on the WDISignal interface.

Note: See "HelloGateway: Sample Application that Handles Application and Gateway Events" for more information.

Outbound Signals – Application Events

These signals occur in response to application events. They are dispatched from the MetaSolv Solution client to your application. For the duration of processing of these signals, the client behaves like a client of your application, invoking operations in the same manner as your application might interact with the MetaSolv Solution Application Server.

Application events have significant differences from gateway events. See "Understanding Events" for more information. Unlike gateway events, application events have no uniform data payload structure and the operations involved differ from one API to the next. In order to handle application events, your application must essentially mimic the behavior of the APIs in terms of implementing the WDIRoot, WDIManager and APINameSession interfaces.

Table 2–9 lists the operations related to outbound application events.

Table 2–9 Operations Related to Outbound Application Events

Interface	Operations	Remarks
WDIRoot	connect	MetaSolv Solution Application Server requests a connection.
WDIRoot	disconnect	MetaSolv Solution Application Server requests to destroy connection.
WDIManager	startAPINameSession	MetaSolv Solution Application Server indicates start of signal. Your application generates an instance of the session interface.
WDIManager	destroyAPINameSession	MetaSolv Solution Application Server indicates end of signal.
APINameSession	As required by the application event	No remarks.

The operations that your application needs to implement for the APINameSession interface will depend on the definition of the application event by the MetaSolv Solution Application Server. This will be a subset of the operations defined in the API's IDL file. In certain cases, the session interface implementation may need to provide operations to generate object references to sub-session interfaces. The sub-session interfaces would then support the lowest level operations.

The following list identifies the main steps in implementing outbound signals for application events:

1. Write server mainline. See "CORBA Client/Server Pattern" for the sample server mainline code.
2. Implement WDIRoot. See "CORBA Client/Server Pattern" for the sample code.
3. Implement WDIManager interface. See "Example Code for Implementing WDIManager and WDISignal Interfaces" for the sample code.
4. Implement *APIName*Session interface and operations as determined by the definition of the application event.
5. Implement sub-session interfaces, if required.

Note: External applications are not required to perform any status updates for application events.

Outbound signals for application events can be handled in the same manner as described in the code sample shown in "Example Code for Updating Status of Events".

Note: Although the steps described above cover the immediate task of handling application event signals sent by the MetaSolv Solution Application Server, you should be aware that the processing of application events usually has a broader scope that extends beyond the signal-handling scenario. Typically, external applications will receive deferred notifications from other external systems (for example, NPAC SMS) that need to be communicated to MetaSolv Solution through the MetaSolv Solution Application Server.

Inbound Signals

MetaSolv Solution enables you to define gateway events as inbound. Inbound signals are the means by which the status of such gateway events may be updated by your application. In contrast to the implementation for outbound signals, where your application is a CORBA server, the implementation for handling inbound signals follows the CORBA client pattern.

The data payload carried by inbound signals is defined in WDI.IDL, reproduced below.

```
struct WDIInEvent
{
    string gatewayName;
    string eventName;
    long   documentNumber;
    long   servItemID;
    char   updateMany;
    string userID;
};
```

The signaling operations are defined in the WDIInSignal interface in WDI.IDL, reproduced below.

```
interface WDIInSignal
{
    WDIStatus eventInProgress(in WDITransaction aWDITransaction,
        in WDIInEvent aWDIInEvent);
    WDIStatus   eventCompleted(in WDITransaction aWDITransaction,
        in WDIInEvent aWDIInEvent);

WDIStatus   eventErrored(in WDITransaction aWDITransaction,
        in WDIInEvent aWDIInEvent, in WDIErrSeq aWDIErrSeq);
};
```

Your implementation for handling inbound signals is no different from the invocation of a synchronous operation on the MetaSolv Solution Application Server.

Error Handling Pattern

This section describes the error handling pattern.

Purpose

The APIs ensure that no data changes applied to the MetaSolv Solution database violate any of the business rules. By the same token, your application should incorporate a robust error-handling scheme to ensure it is in sync MetaSolv Solution with regard to the status of operations and the state of data in the MetaSolv Solution database.

When Used

Any time an operation is invoked, whether the operation is invoked by your application or by MetaSolv Solution, error handling is involved. Specifically, your application should:

- Ensure that the status of all operations invoked on the APIs is captured and examined for errors. This minimizes the possibility of these errors being propagated downstream.
- Ensure that meaningful status information is returned to the API in those instances where the API initiates the interaction with your application.

Description

The data structures used to communicate the status of operations are defined in IDL file WDI.IDL. These are:

- WDIExcp: Exception
- WDIErrSeq: Error Array
- WDIStatus: Status

These are reproduced in the following paragraphs.

Exception

```
exception WDIExcp
{
    long   code;
    string reason;
```

```
};
```

Exceptions are the most commonly employed mechanism to indicate errors. The WDIExcp exception object contains an error code and error description. Exceptions are used in the following scenarios:

- All operations on WDIRoot, WDIManager and WDITransaction interfaces
- Most operations on *APIName*Session interface, except WDIPSRSession

The following code fragment shows how exceptions may be caught.

```
try {
    aWDIManager = aWDIRoot.connect(req);
}
catch (WDIExcp ex) {
    System.err.println("connect failed [" + ex.code + "]: " + ex.reason);
}
```

Error Array

```
struct WDIError
{
    long   code;
    string reason;
};
typedef sequence<WDIError> WDIErrSeq;
```

The error array WDIErrSeq is defined as an array of error objects of type WDIError. Each element of the array contains an error code and error description. The error array is also contained in the status object WDIStatus. The error array is used to communicate errors in the following scenarios:

- For callback operations on the WDINotify interface that indicate failure
- When your application sets the status of gateway events to Errored

The following Java code fragment shows an example of how the error array is used by the APIs. The DLRGetFailed_v5 operation gets called back by DLRSERVER to indicate failure of the getDLR_v5 operation.

```
public void DLRGetFailed_v5(MetaSolv.CORBA.WDIDLRTypes_v5.DLRRequest aDLRRequest,
                    MetaSolv.CORBA.WDI.WDIError[] aWDIErrSeq)
  {
  // In practice, you may want to persist the error array somewhere
  // and indicate operation failure to the interested parties.
  //
  // This code displays all error messages on the console
  System.err.println("getDLR()_v5 failed. Errors returned by server:");
  for (int I=0; i<aWDIErrSeq.length; i++) {
    System.err.println("\tReason: " + aWDIErrSeq[i].reason +
        " [Code: " + aWDIErrSeq[i].code + "]");
  }
```

Status

```
struct WDIStatus
{
    boolean   aResult;
    WDIErrSeq aWDIErrSeq;
};
```

The WDIStatus object defines operation status. This object comprises a Boolean element aResult that indicates success or failure and an error vector aWDIErrSeq. The status object is used in the following scenarios:

- All inbound signaling operations (the operations on WDIInSignal)
- Outbound signaling operations with the exception of *eventOccurred* and *eventTerminated*

This concept is illustrated with the following code fragment that shows use of WDIStatus in signaling. The code fragment shows how an external application that updates status of gateway events might handle error status returned by the MetaSolv Solution Application Server.

```
try {
    WDIStatus myStatus = aWDISignal.eventInProgress(aWDITransaction, aWDIEvent);
    if (myStatus.aResult == true) {  // operation successful
        aWDITransaction.commit();
    }
    else {   // operation failed
        aWDITransaction.rollback();
        String msg = "Error updating event status: \n";
        for (int i=0; I<myStatus.aWDIErrSeq.length; i++) {
            msg = msg + "\tReason: " + myStatus.aWDIErrSeq[i].reason +
            " [Code: " + myStatus.aWDIErrSeq[i].code + "]\n";
        System.err.println(msg);
```

It is your application's responsibility to capture, log and interpret all errors received from MetaSolv Solution. See "API Error Messages and Exceptions" for more information about error messages that can be returned from the APIs.

If required, the API administrator can configure the MetaSolv Solution Application Server to send notifications of specific errors by e-mail and/or by pager e-mail to one or more e-mail or pager addresses. Instructions for setting up notifications are provided in the *MSS System Administrator's Guide*.

Your applications that communicate errors to MetaSolv Solution must do so through the Error Array and WDIStatus data structures. MetaSolv Solution records and display these errors for the user, but cannot interpret or act on third-party error messages directly.

Sample Applications

The following sections discuss two sample applications.

HelloAPI: Sample Application that Exports Data

This section describes the development of a simple application, HelloAPI.

HelloAPI builds upon several of the implementation patterns discussed earlier in this chapter:

- Basic API Setup Pattern
- Asynchronous Interaction Pattern
- CORBA Client/Server Pattern
- Error Handling Pattern

HelloAPI invokes the Inventory and Capacity Management (ICM) API to perform a simple data export operation.

> **Note:** Once compiled for the customer's environment, an application of this sort may be used as a starter template for building applications that interact with the APIs. Such an application may also be useful as a simple diagnostic tool to determine whether the development, test, or production environment is set up and configured correctly and has all required applications running.

The HelloAPI consists of two files. The first file, HelloAPI.java, contains Java code that performs mainline functions for the sample application. This application:

- Sets up a connection with DLRSERVER
- Invokes a data export operation on DLRSERVER
- Waits on a callback from DLRSERVER that returns the results of the export operation that was requested
- Displays the results on the console, then exits

The second file, WDIExampleNotificationImpl.java, contains Java code that implements the callback interface required to support the asynchronous export operation performed in the server mainline code.

The output from the application should resemble the sample below; however, actual data values may vary.

```
Sent request to DLRSERVER. Waiting on notify callback ...
Circuit ECCKT:  /HCG-/000026/   /MGCM/
Exiting application
```

Implementation Notes

This example uses the object access serialization mechanisms supported by the Java language to suspend thread activity pending callback invocation from the API server. Alternate synchronization mechanisms that achieve the same effect can be used instead.

In the scenario used for this sample, the HelloAPI application is not registered with the ORB and the application's host machine does not run the ORB daemon.

The code for this sample application can be modified to invoke operations on other API. However, when invoking asynchronous operations on other APIs, the WDINotify interface that is implemented must be the one defined in that specific API's IDL file.

The HelloGateway sample application consists of these Java files:

- HelloAPI.java contains the application's mainline code.
- WDIExampleNotificationImpl.java implements callback interfaces.
- Utils.java contains the code that connects to the ORB.

HelloGateway: Sample Application that Handles Application and Gateway Events

This section describes the development of a Java application, HelloGateway, that handles signals originated by MetaSolv Solution clients when gateway events occur.

HelloGateway builds upon all of the implementation patterns discussed earlier in this chapter:

- Basic API Setup Pattern
- Synchronous Interaction Pattern
- Asynchronous Interaction Pattern
- CORBA Client/Server Pattern
- Signal Handling Pattern
- Error Handling Pattern

The HelloGateway sample application receives gateway events from MetaSolv Solution clients. For the purpose of this sample, the application does not perform any external processing based on the event. The application only receives the event, then updates the event status to In Progress and then to Completed.

The HelloGateway application consists of these Java files:

- HelloGatewayServer.java contains the server's mainline code.
- WDIGatewayRootImpl.java implements the WDIRoot interface.
- WDIGatewayManagerImpl.java implements the WDIManager interface.
- WDIGatewaySignalImpl.java implements the WDISignal interface.
- RequestThread.java implements a Java thread that processes a single event. All the thread does is update the event status.
- SendSignal.java sends gateway event signals to the MetaSolv Solution Application Server.
- Utils.java contains the code that connects to the ORB.

Migrating to MetaSolv Solution 6.3.*x* from 6.2.x

The MetaSolv Solution 6.3.*x* CORBA APIs use JacORB 3.8. If you are migrating from MetaSolv Solution 6.2.*x*, you must migrate the CORBA APIs to use this version of JacORB.

If you are using your own independent ORB, you do not need to migrate the CORBA APIs to use JacORB, because the client code depends on the ORB with which it is communicating.

To migrate the CORBA APIs to use JacORB 3.8:

1. Install the following:
   - JacORB 3.8 (provided with the MSS 6.3.*x* installation)
   - JDK 8 (with the latest critical patches), you need to download and install this separately
2. Remove the following JacORB JAR files from your classpath:
   - jacorb.jar (provided with an older JacORB version)
   - jacorb_stubs.jar (provided with Metasolv Solution 6.2.x)
   - slf4j-api-*version*.jar
   - slf4j-jdk14-*version*.jar
   - backport-util-concurrent.jar

where *version* is the existing version in the filename.

3. Add the following JacORB JAR files to your classpath:
   - jacorb.jar (provided with JacORB 3.8)
   - jacorb-omgapi-3.8.jar
   - jacorb-services-3.8.jar
   - jacorb_stubs.jar (provided with MetaSolv Solution 6.3.*x*)
   - slf4j-api-1.7.14.jar
   - slf4j-jdk14-1.7.14.jar

   The **jacorb_stubs.jar** file is part of the MetaSolv Solution installation. The remaining JAR files are located in the *JacORB_home***/lib** directory, where *JacORB_home* is the directory of where JacORB is installed.

Migrating to MetaSolv Solution 6.3.*x* from 5.x and 6.0.x

MetaSolv Solution releases before 6.2.x, CORBA APIs used JBroker. The MetaSolv Solution 6.3.*x* CORBA APIs use JacORB. If you are migrating from a previous release of MetaSolv Solution, you may need to migrate the CORBA APIs to use JacORB, and configure JacORB:

- If you are using the JBroker ORB that was provided with the MSS version prior to 6.2.0, you must migrate the CORBA APIs to use JacORB, and configure JacORB, as described in the following procedures.
- If you are using your own independent ORB, you do not need to migrate the CORBA APIs to use JacORB, or configure JacORB, because the client code depends on the ORB with which it is communicating.

To migrate the CORBA APIs to use JacORB:

1. Install the following:
   - JacORB 3.8 (provided with the MSS 6.3.*x* installation)
   - JDK 8 (with the latest critical patches), you need to download and install this separately
2. Remove the following JBroker JAR files from your classpath:
   - jbroker_stubs.jar
   - jbroker-rt.jar
   - jbroker-ssl.jar
   - jbroker-tools.jar
   - log4j.jar
3. Add the following JacORB JAR files to your classpath:
   - jacorb.jar
   - jacorb-omgapi.jar
   - jacorb-services.jar
   - jacorb_stubs.jar
   - slf4j-api-1.7.14.jar

- slf4j-jdk14-1.7.14.jar

The **jacorb_stubs.jar** file is part of the MetaSolv Solution installation. The remaining JAR files are located in the *JacORB_home*/**lib** directory.

4. Pass the **jacorb.home** value as a virtual memory (VM) to the class where the integration code resides. For example, the following shows the **jacorb.home** value to be the directory where JacORB is installed:

```
-Djacorb.home=C:\MSS_Domains\BEA1213_WLS_SSL\AdminServer\jacORB
```

5. When initializing the orb, pass the **org.omg.CORBA.ORBClass** property value as *org.jacorb.orb.ORB* instead of *com.sssw.jbroker.ORB*. For example:

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass", "org.jacorb.orb.ORB");
orb = ORB.init(new String[0], props);
```

If you install the MSS 6.3.*x* application in an SSL-enabled environment, you must also enable SSL in JacORB. See "Implementing SSL in JacORB 3.8" for more information.

6. When retrieving any CORBA object reference, convert the reference object to the corresponding object type using its helper class. For example:

```
TestWDINotification iNotif;
WDINotification aTestWDINotification = null;
```

This works the same way with JBroker. For example:

```
iNotif = new TestWDINotification();
org.omg.CORBA.Object ref = this.connectObject(iNotif);
aTestWDINotification =
(WDINotification)WDINotificationHelper.narrow(ref);
```

In addition to migrating the CORBA APIs to JacORB, you also need to configure JacORB.

To configure JacORB:

1. Edit the **orb.properties** file, located in the *JacORB_home*/**etc** directory. Provide appropriate values for the following properties:

> **Note:** If any of these properties are commented, uncomment them.

   a. Set the **jacorb.config.dir** property value to the JacORB home directory. This is generally *<domain name>*/*<server name>*/jacORB. For example:

```
//jacorb.config.dir=<Root directory of JacORB under the domain>
jacorb.config.dir= C:\MSS_Domains\MSS_HOME_BEA_MIGRATE\AdminServer\jacORB
```

   This property enables the root directory of the JacORB.

   b. Set the **jacorb.config.log.verbosity** property value to 4 for a testing environment. Otherwise set the value to 1 (errors) or 2 (warnings).

   c. Set the **jacorb.log.default.verbosity** property value to 4 for a testing environment. Otherwise, set the value to 1 (errors) or 2 (warnings).

   d. Comment the **jacorb.naming.ior_filename** property.

e. Set all the remaining log property values to 4 for a testing environment. Otherwise set the values 1 (errors) or 2 (warnings).

2. Edit the **jacorb.properties** file, located in the *JacORB_home*/**etc** directory. Provide appropriate values for the following properties:

Note: If any of these properties are commented, uncomment them.

a. Set the **ORBInitRef.NameService** property value to where the NameService.ior is going to be available.

For example, the value can be a file on a shared drive:

```
file:/C:/MSS_Domains/MSS_HOME_BEA_
MIGRATE/AdminServer/appserver/ior/NameService.ior
```

For another example, the value can be a file available at a host and port:

```
http://192.0.2.66:15000/NameService
```

The latter example works only if the **URLNamingServicePort** property is enabled and has a value in the **gateway.ini** file.

b. Set the **jacorb.log.default.verbosity** property value to 4 for a testing environment. Otherwise, set the value to 1 (errors) or 2 (warnings).

c. Set all the remaining log property values to 4 for a testing environment. Otherwise, set the values to 1 (errors) or 2 (warnings).

d. Set the **jacorb.naming.ior_filename** property value to where the NameService.ior needs to be generated.

Typically, this is the JacORB home directory, which is generally *domain name*/*servername*/**appserver/ior** folder. For example:

```
C:/MSS_Domains/MSS_HOME_BEA_
MIGRATE/AdminServer/appserver/ior/NameService.ior
```

This property tells the Name Service where to generate the IOR file. The value of this property can be a file name and location path, or a URL. If the name server is in a shared location, set the value to a URL (a logical location exists in your domain, where the application server is running). Otherwise, set the value to the file name and location path.

3. Copy the **orb.properties** file from *the MSLV_Home**mslv01***jacORB\etc** directory to the *MSLV_Home**mslv01***jacORB** directory, where *MSLV_Home* is the directory in which the MSS software is installed and *mslv01* is the server home directory.

If SSL is enabled on JacORB, ensure that you provide the appropriate values for the following properties in the *MSLV_Home**mslv01***jacORB\orb.properties** file after you install and before you deploy the MSS application:

```
jacorb.security.keystore=C:/MSSSSL/identity.jks
jacorb.security.keystore_password=password
jacorb.security.default_user=mycert
jacorb.security.default_password=password
```

Implementing SSL in JacORB 3.8

To enable SSL in JacORB, ensure that you provide the appropriate values for the following properties in the *MSLV_Home**mslv01***jacORB\orb.properties** file after you install and before you deploy the MSS application:

```
jacorb.security.keystore=C:/MSSSSL/identity.jks
jacorb.security.keystore_password=password
jacorb.security.default_user=mycert
jacorb.security.default_password=password
```

In the integration code, when instantiating the ORB instance, you must set the following properties if SSL is enabled in JacORB.

```
Properties props = new Properties();
//Add the following properties to the ORB if SSL is enabled on JacORB.
props.put("jacorb.security.support_ssl", "on");
props.put("jacorb.ssl.socket_factory", "org.jacorb.security.ssl.sun_
jsse.SSLSocketFactory");
props.put("jacorb.ssl.server_socket_factory", "org.jacorb.security.ssl.sun_
jsse.SSLServerSocketFactory");
props.put("jacorb.security.keystore", "C:/MSSSSL/identity.jks"); //Location of
//the private keystore.
props.put("jacorb.security.keystore_user", "mycert"); //Private keystore alias
//name.
props.put("jacorb.security.keystore_password", "password"); //Private keystore
//password.
props.put("jacorb.security.jsse.trustees_from_ks", "on");
props.put("jacorb.security.truststore", "C:/MSSSSL/trust.jks"); //Location of
//the public keystore.
props.put("jacorb.security.truststore_user", "mycert"); //Public keystore alias
//name.
props.put("jacorb.security.truststore_password", "password"); //Public keystore
//password.
props.put("jacorb.security.ssl.client.supported_options", "20");
props.put("jacorb.security.ssl.client.required_options", "20");
props.put("jacorb.security.ssl.server.supported_options", "20");
props.put("jacorb.security.ssl.server.required_options", "20");
//General properties
props.put("org.omg.CORBA.ORBClass", "org.jacorb.orb.ORB");
//ORB instance creation
orb = ORB.init(new String[0], props);
```

You use Java KeyStore (.jks) to store security certificates, including private and public keys.

The jacorb.security.keystore property looks for a private key containing a self-signed digital certificate. This property is also known as identity key store.

The jacorb.security.truststore property looks for a key store containing a trusted certificate authority (CA) certificate. This property is also known as public key.

For information about public and private key stores and how to create and configure them in the WebLogic Administration Console, see the following website:

http://docs.oracle.com/cd/E24329_01/web.1211/e24422/identity_trust.htm

For more information, see "IIOP over SSL" in the JacORB Programming Guide 3.8 at:

http://www.jacorb.org/documentation.html

3

Common Architecture

Figure 3–1 shows the common interfaces of the Oracle Communications MetaSolv Solution APIs.

Figure 3–1 Common IDL Architecture Interfaces

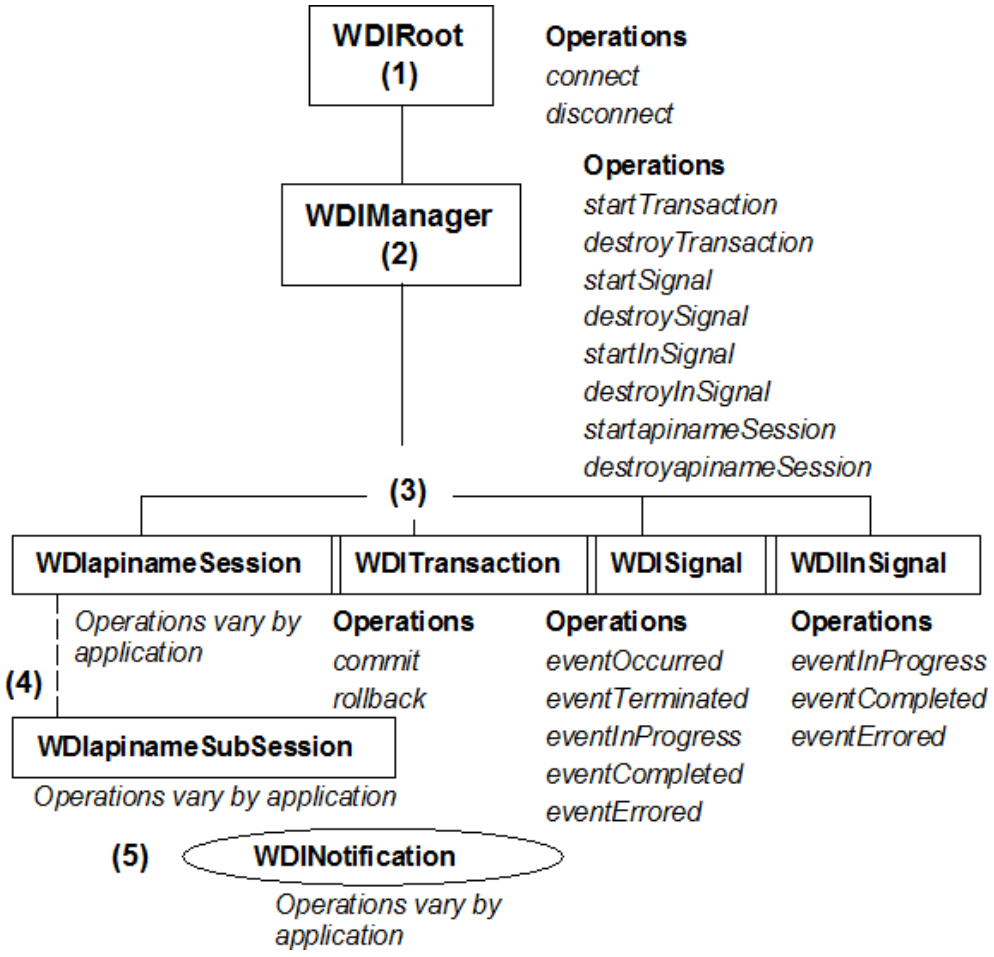


Table 3–1 lists the keys for common IDL architecture interfaces.

Table 3–1 Keys for Common IDL Architecture Interfaces

Key	Architecture Description
(1)	At the highest level is the WDIRoot interface, which enables connection management. The WDIRoot layer provides services that are used throughout the architecture and serve as the connection factory. See "WDIRoot Interface" for more information.
(2)	The second level is the WDIManager interface, which enables session, signal, and transaction management. The WDIManager object reference is obtained from a successful *connect* to the WDIRoot. The connection operation of the WDIRoot object returns an object reference to a WDIManager. WDIManager provides services to start and destroy transaction objects, signal objects, and session objects.
(3)	The third level contains the session, signal, and transaction interfaces, whose object references are obtained from the WDIManager interface. See Figure 3–4, "WDIManager Interface" for more information.
(4)	The optional fourth level contains the more granular subSession object whose object reference is obtained from the parent session interface. Refer to the sample flows presented in each API chapter for examples.
(5)	The API architecture defines a callback mechanism that is exposed by the WDINotification interface.

The session or subSession enables access to business application operations. These operations are detailed business objects that vary by the business functions exposed.

Note: MetaSolv Solution APIs do not necessarily require every interface defined in this document.

WDIRoot Interface

The *connect* operation of the WDIRoot interface obtains the object reference to the WDIManager.

Figure 3–2 shows the WDIRoot interface.

Figure 3–2 WDIRoot Interface

Connection to the MetaSolv Solution Application Server

To begin a connection, the third-party application must connect to the MetaSolv Solution Application Server. This connection verifies the user ID and password, and returns the object reference to the APIs WDIRoot. The connection operation returns a reference to a WDIManager object.

Connection to the CORBA Daemon

By default, all APIs perform an **impl is ready** connection to the daemon in order to register the availability of its object references. The MetaSolv Solution API system administrator can set the StrictOMG system parameter to **true** in the MetaSolv Solution Application Server INI file. The result is an OMG ORB connect on the WDIRoot object.

After performing the OMG ORB connection, the application server writes an OMG stringified object reference for the WDIRoot object to a file, using the file name specified by the IORPath system parameter in the application server INI file and the name of the API server.

Connection to the Root Object

Figure 3–3 illustrates the connection process.

Figure 3–3 Connection Process

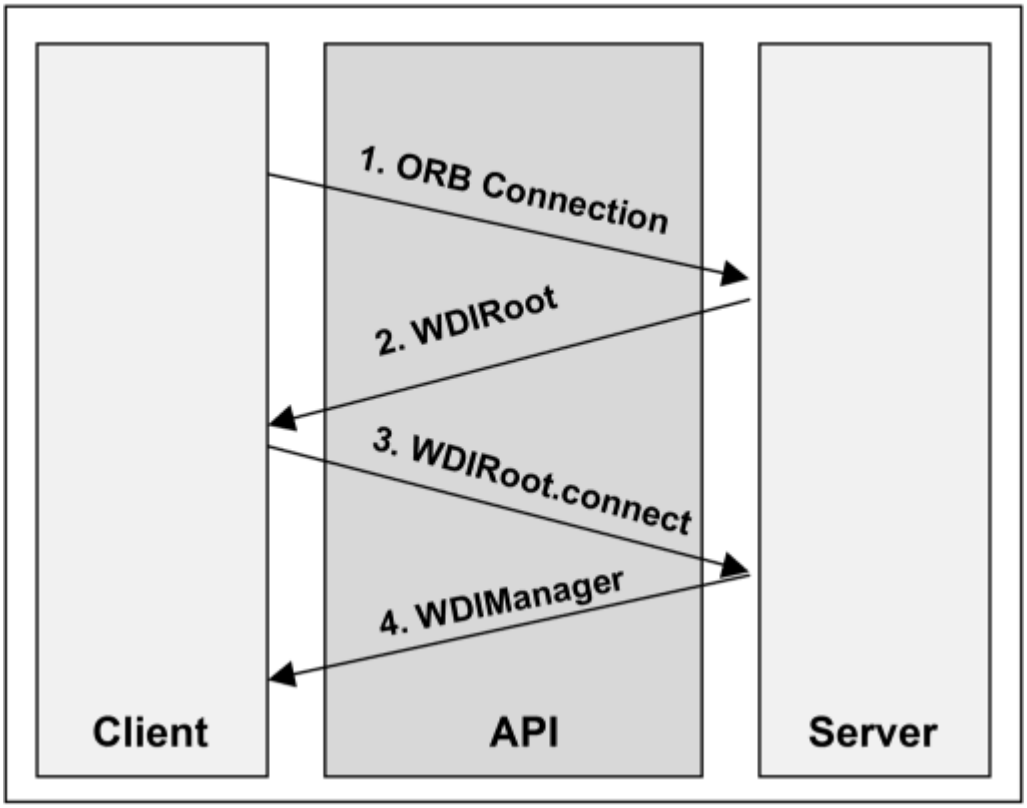


Table 3–2 lists the operations exposed by the WDIRoot interface.

Table 3–2 WDIRoot Interface Operations

Operation Name	Description
connect	Returns a reference to WDIManager
disconnect	Terminates the connection

WDIManager Interface

The object reference to the WDIManager is obtained by initiating the *connect* operation of the WDIRoot interface, as shown in Figure 3–4.

Figure 3–4 WDIManager Interface

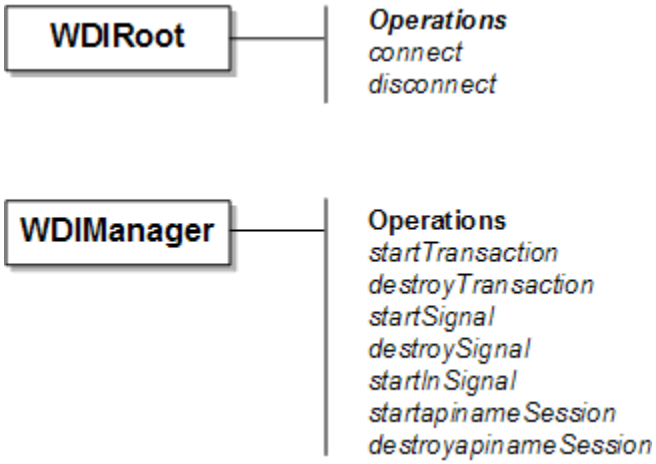


Table 3–3 lists the operations exposed by the WDIManager interface.

Table 3–3 WDIManager Interface Operations

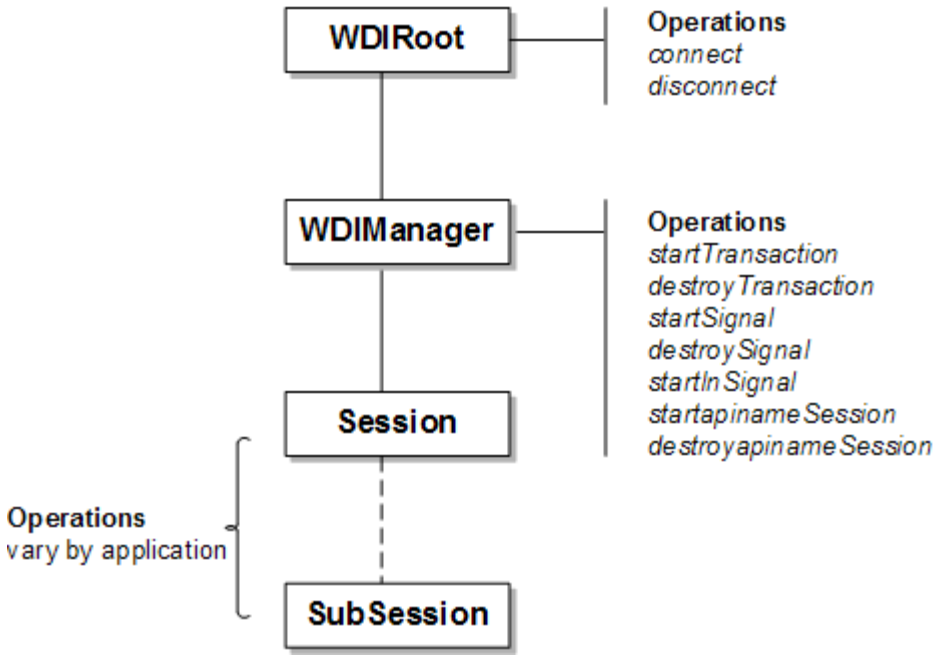
Operation	Description
startapinameSession	Obtains the object reference of the *apinameSession* where *apiname* designates the specific API, as in *startLSRSession*
destroyapinameSession	Terminates the established session, as in *destroyLSRSession*
startTransaction	Establishes a database connection using this process: The *Start* operation makes the connection and establishes a database transaction object. The API returns a handle for that connection to the initiating process. The term, handle, is synonymous with a WDITransaction object reference.
destroyTransaction	Invalidates a database transaction object. Any pending changes are lost when this function is called if it was not preceded by a *commit*.
startSignal	Obtains the WDISignal object reference
destroySignal	Terminates the Signal
startInSignal	Obtains the WDI Insignal object reference
destroyInSignal	Terminates the Insignal

Note: Some APIs do not define *startSignal*, *destroySignal*, *startInSignal*, *destroyInSignal*, *startTransaction*, or *destroyTransaction*. For details about a specific API, see the chapter of this guide that describes that API.

API Session Interfaces (Session Processing)

The object reference to the *apiname*Session is obtained by initiating the start*apiname*Session operation of the WDIManager, as shown in Figure 3–5.

Figure 3–5 Basic Session Interface



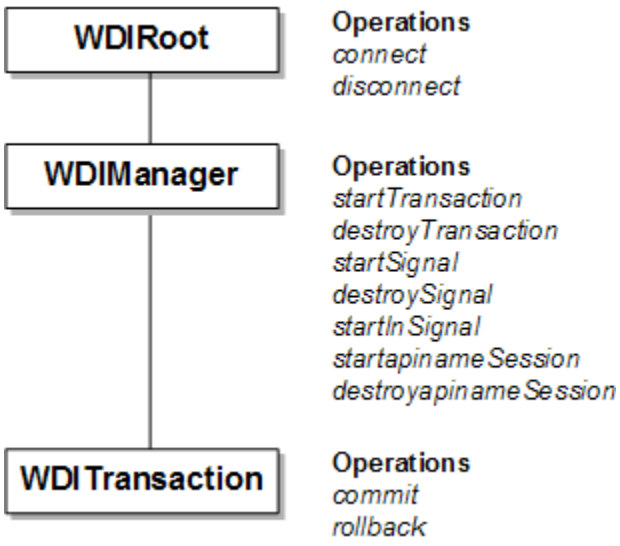
Operations in sessions and subsessions vary according to the API.

WDITransaction Interface (Database Transactions)

The object reference to the WDITransaction is obtained by initiating the *startTransaction* operation of the WDIManager interface. Third-parties can use the API to coordinate the database transactions because no assumed paths or commit points are built into the API. *Commit* and *rollback* are operations of the WDITransaction interface.

Figure 3–6 shows the WDITransaction interface.

Figure 3–6 WDITransaction Interface



The *commit* operation uses a database handle and saves any pending changes to the database. Once a commit has occurred, all database updates are applied to the database. After a commit, the transaction object is still valid and can continue to be used.

The *rollback* operation uses a database handle and rolls back any pending changes to the database. Once a rollback has occurred, any pending database changes are discarded. After a rollback, the transaction object is no longer valid and cannot be used for further operations.

In APIs that use the commit and rollback operations, your application must specifically call commit and rollback. However, some APIs do not use the WDITransaction interface. In these cases, MetaSolv Solution is responsible for database transaction management.

WDISignal Interface (Outbound Signal Processing)

The object reference to the WDISignal is obtained by initiating the *startSignal* operation of the WDIManager interface.

If the signal is a gateway event signal, certain key data as defined in the WDIEvent structure (in the IDL) is passed. If the signal is an application event signal, the data to be passed varies, depending on the application.

The third party is responsible for implementing the *eventOccurred* and *eventTerminated* operations of the WDISignal interface. MetaSolv Solution is responsible for implementing the remaining operations of the WDISignal interface.

Figure 3–7 shows the WDISignal interfaces.

Figure 3–7 WDISignal Interfaces

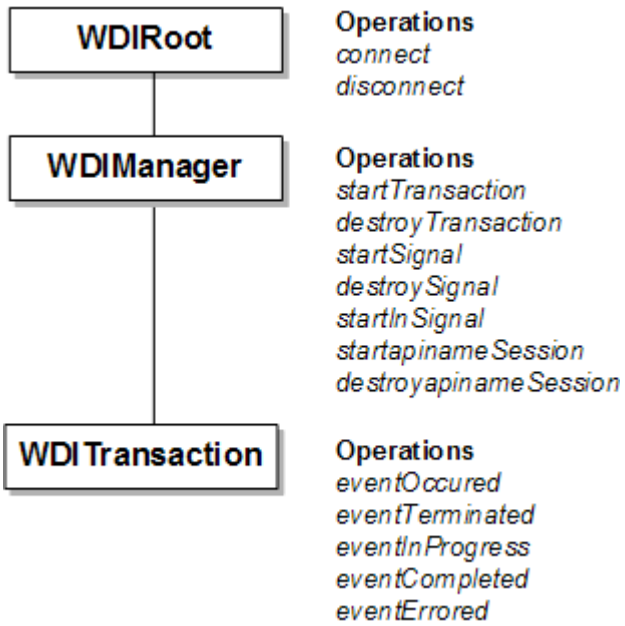


Table 3–4 lists the operations exposed by the WDISignal interface.

Table 3–4 WDISignal Interface Operations

Operation	Description
eventOccurred	MetaSolv Solution initiates a signal indicating that a gateway event of interest to the third-party software has occurred.
eventTerminated	MetaSolv Solution terminates an event to notify the third-party application that MetaSolv Solution is no longer interested in completing the event and is no longer interested in receiving status updates for the event.
eventInProgress	The third-party application sets the status of the gateway event to In Progress when it has received and begun processing the signal.
eventCompleted	The third-party application sets the status of a gateway event to Completed when it has successfully finished processing the event.
eventErrored	The third-party application sets the status of the gateway event to Error when an error has occurred while processing the event. This operation also provides a mechanism for error information to be communicated to the API.

WDIInSignal Interface (Inbound Signal Processing)

The object reference to the WDIInSignal is obtained by initiating the *startInSignal* operation of the WDIManager interface. The WDIInSignal interface allows the third-party application to update statuses of unsolicited or inbound gateway events in the Work Management subsystem. This is illustrated in Figure 3–8.

Figure 3–8 WDIInSignal Interfaces

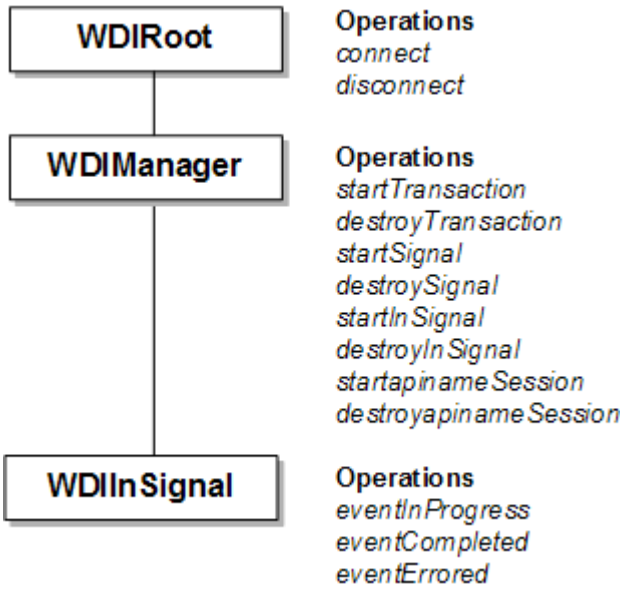


Table 3–5 lists the operations exposed by the WDIInSignal interface.

Table 3–5 WDIInSignal Interface Operations

Operation	Description
eventInProgress	The third-party application sets the status of a gateway event to In Progress when it has started processing an event.

Table 3–5 (Cont.) WDIInSignal Interface Operations

Operation	Description
eventCompleted	The third-party application sets the status of a gateway event to Completed when it has successfully finished processing an event.
eventErrored	The third-party application sets the status of a gateway event to Error when an error has occurred while processing an event.

WDINotification Interface (Callback Mechanism)

Most operations implemented within the APIs require a WDINotification object reference as the first input parameter. The third-party application instantiates a WDINotification object.

The WDINotification interface enables a callback mechanism to notify the third-party application of the result of an operation invoked against it. The callback mechanism is used to communicate the results of an asychronous operation.

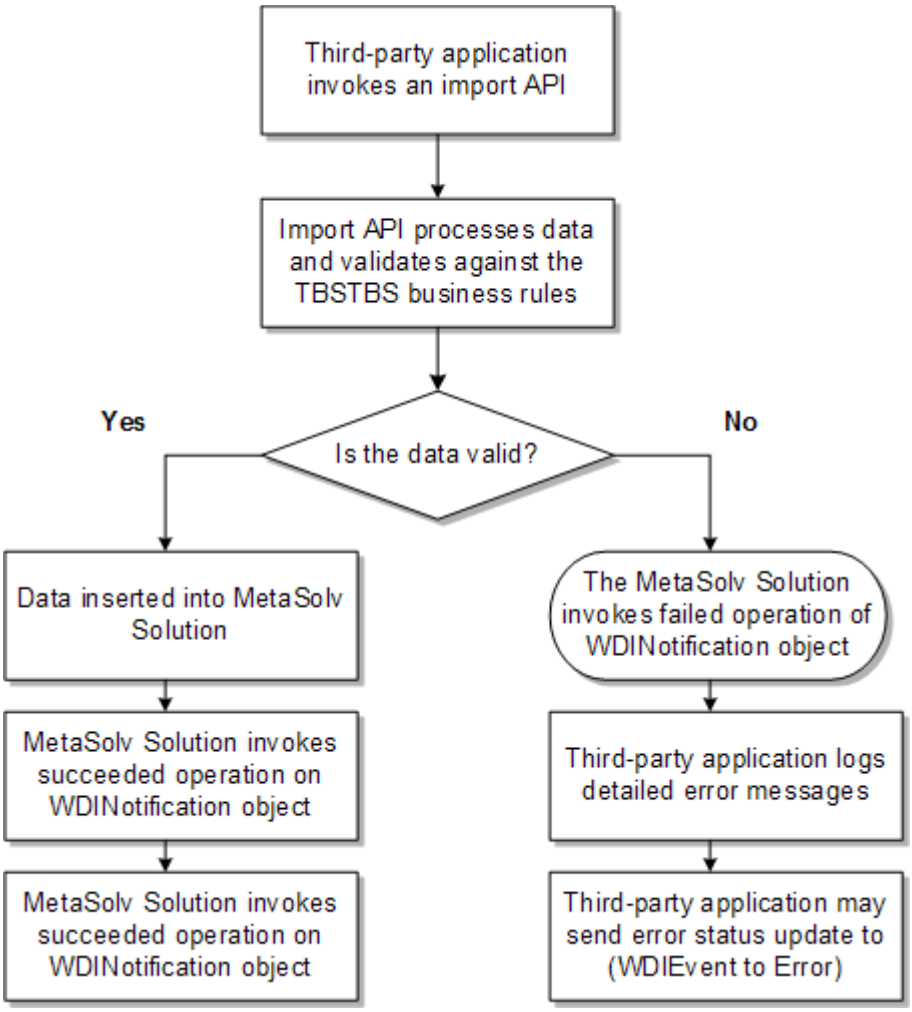
Note: Operations of the WDINotification interface vary according to the application.

MetaSolv Solution uses JacOrb 3.8 to support the CORBA API. This software supports the 2.4 CORBA Standard. This software is shipped as part of the MetaSolv Solution product line and does not require you to purchase anything from a third party. This ORB inter operates with all the available ORBs.

The WDINotification interface has operations defined indicating the success or failure of an invoked operation. The parameters of the WDINotification interface include a reference (which varies by application, for example a document number for LSR-related callbacks) and an error structure where appropriate. The third-party application is responsible for implementing the operations in the WDINotification interface.

If an error is encountered during the processing of an API object implementation, a callback is performed to the third-party application indicating that the operation has failed and why the operation failed. The reasons for the failure are communicated by an error structure (WDIErrSeq), which may contain multiple, detailed error messages. Different errors can be encountered while attempting to process a given request, as shown in Figure 3–9.

Figure 3–9 Sample Flow for Successful and Error Conditions



Displaying errors is the responsibility of the third-party application because the third-party application functionality and error processing differs among software package. The API provides a mechanism for errors to be communicated back to the Work Management subsystem through the *eventErrored* operation of the WDISignal interface.

For example, with a third-party application importing a local service request confirmation (LSC) to the LSR API, the following processes can occur:

1. The third-party application invokes the *importLSC* operation with the appropriate data.
2. The *importLSC* operation processes the LSC data.
   - During the processing, the LSC is validated against the MetaSolv Solution database business rules for the LSC.
   - If all data is valid, it is inserted into the MetaSolv Solution database, and the successful operation is invoked on the notification interface.
   - No detail error messages are generated.

If the import LSC process encounters an error or errors, the LSC is not inserted into the MetaSolv Solution database. The error code structure is populated with detailed error message information. Examples of these include **location code not in the database** and

NC/NCI codes are invalid. MetaSolv Solution invokes the failed operation of the WDINotification interface.

The third-party application is responsible for logging the errors and making them available to the user. Typically, this means the data displays to the user. The third-party application may optionally communicate this error information back to MetaSolv Solution through the error status update for an event.

4

The Infrastructure API

Much of the underlying information in the database is managed by the Oracle Communications MetaSolv Solution Infrastructure subsystem.

Specific operations for exporting lists of information from the database are provided by the Infrastructure API. These lists of information include:

- Structured formats and structured format components
- Geographic areas and types
- Code categories and code category values, including languages
- Network locations

Additional operations are provided and used to manage end-user and network location information. See "NetworkLocationSubSession" for more information.

The CORBA server name used by the Infrastructure API is INFRASTRUCTURESERVER.

Implementation Concepts

This section describes the implementation concepts for the Infrastructure API.

Infrastructure Operational Differences

This section describes the operational differences between the Infrastructure subsystem and the API.

Latitude and Longitude Fields Are Not Calculated and Validated

Unlike the Infrastructure subsystem in MetaSolv Solution, the Infrastructure API does not allow for calculation of the **Latitude** and **Longitude** fields if data is entered in the **Vertical** and **Horizontal** fields of the *associateLocationRelationships* operation. Validation also does not occur if data is entered into the **Latitude** and **Longitude** fields.

Switch Network Area Field Defaults to First Switch Network Area

The Infrastructure API does not support the **Switch Network Area** field selection in the *queryNetworkLocations_V2* operation.

If there is only one switch network area in the **Switch Network Area** field, the **Switch Network Area** field is defaulted to it. If there are more than one switch network areas in the **Switch Network Area** field, the first switch network area listed in the database, populates that field.

Query Across All Address Formats

Unlike the Infrastructure subsystem in MetaSolv Solution, the Infrastructure API does not allow you to query for all address formats when using the *queryNetworkLocation_V2* operation.

Key MetaSolv Solution Concepts

To understand the information made available through the Infrastructure API, you must understand certain key concepts used in MetaSolv Solution. In particular, you should understand how MetaSolv Solution uses these kinds of information:

- Code categories and code category values
- Geographic areas and types
- Network locations
- Structured formats and structured format components
- Customized attributes (CAs) - CAs are what MetaSolv Solution users use to add an attribute (or property, or value) to a building block. They offer a way to add company-specific information to MetaSolv Solution. They are custom because your company's unique business processes and technological practices dictate how CAs are used. Several CAs are included in the data that comes with MetaSolv Solution, and those CAs are immediately available for association to building blocks. MetaSolv Solution users can also create new CAs.

  Building blocks are the only parts of the software with which CAs can be associated. Templates, elements, connections, and connection allocations are the four building block types from which MetaSolv Solution users can select building blocks for association with a CA. Building block types are predefined and unchangeable.

Infrastructure API Files

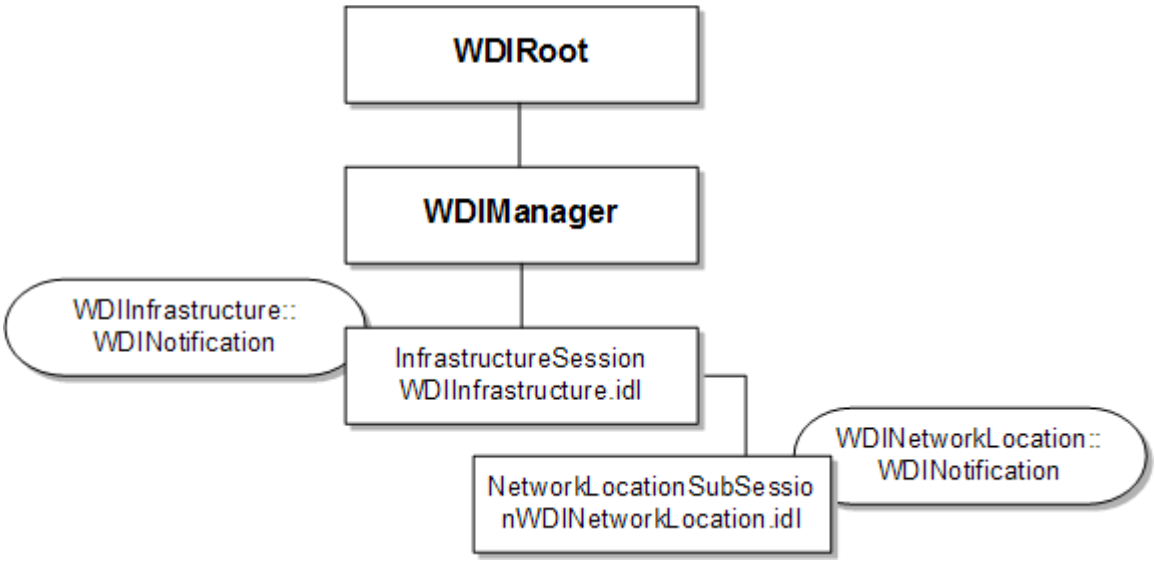
These IDL files are used in the Infrastructure API:

- WDIInfrastructure.idl
- WDIInfrastructureTypes.idl
- WDINetworkLocation.idl
- WDINetworkLocationTypes.idl
- WDINetworkLocationTypes_v2.idl
- WDI.idl
- WDIUtil.idl

Infrastructure Interface

Figure 4–1 shows the relationship of the interfaces within the Infrastructure API.

Figure 4–1 Infrastructure API Interfaces



WDIManager

Table 4–1 lists the operations available in the WDIManager interface of the **WDIInfrastructure.idl** file.

Table 4–1 WDIManager Interface Operations in the Infrastructure API

Operation	Description
startInfrastructureSession	Obtains the object reference of the InfrastructureSession
destroyInfrastructureSession	Terminates the InfrastructureSession
startTransaction	commit rollback
destroyTransaction	Terminates the transaction
startNetworkLocationSubSession	Returns the NetworkLocationSubSession
destroyNetworkLocationSubsession	Destroys the NetworkLocationSubsession

Note: See "WDIManager Interface" for more information on the WDIManager interface.

InfrastructureSession Interface

Table 4–2 lists the operations that comprise the InfrastructureSession in the **WDIInfrastructure.idl** file.

Table 4–2 Infrastructure API InfrastructureSession Interface Operations

Operation	WDIInfrastructure::WDINotification Operations
getMaximumReturnedRows	Implemented by caller to return maximum number of records for Infrastructure API server to return for certain queries (0 = no limit).
getStructureTypes	getStructureTypesSucceeded operationFailed
getStructureFormatsGivenType	getStructureFormatsGivenTypeSucceeded operationFailed
getStructureFormatsGivenTypeAndArea	getStructureFormatsGivenTypeAndAreaSucceeded operationFailed
getComponentsGivenStructureFormat	getComponentsGivenStructureFormatSucceeded operationFailed
getValidValuesGivenStructureFormat Component	getValidValuesGivenStructureFormatComponent Succeeded operationFailed
getGeoAreaTypes	getGeoAreaTypesSucceeded operationFailed
getGeoAreaTypesGivenCountry	getGeoAreaTypesGivenCountrySucceeded operationFailed
getGeoAreasGivenType	getGeoAreasGivenTypeSucceeded operationFailed
getGeoAreasGivenTypeAndCountry	getGeoAreasGivenTypeAndCountrySucceeded operationFailed
getRelatedGeoAreasGivenAreaAndType	getRelatedGeoAreasGivenAreaAndTypeSucceeded operationFailed
getLanguages	getLanguagesSucceeded operationFailed
getCodeCategories	getCodeCategoriesSucceeded operationFailed
getCodeCategoryValues	getCodeCategoryValuesSucceeded operationFailed
queryConditionCode	queryConditionCodeSucceeded operationFailed
getConditionCode	getConditionCodeSucceeded operationFailed
getNetworkAreasGivenLocation	getNetworkAreasGivenLocationSucceeded operationFailed
startNetworkLocationSubSession	Obtains the object reference of the NetworkLocationSubSession
destroyNetworkLocationSubSession	Terminates the NetworkLocationSubSession

Table 4–2 (Cont.) Infrastructure API InfrastructureSession Interface Operations

Operation	WDIInfrastructure::WDINotification Operations
getCaUsageSetInfoFromServiceItem	getCaUsageSetInfoFromServiceItemSucceeded operationFailed
getCaUsageSetInfoFromTemplate	getCaUsageSetInfoFromTemplateSucceeded operationFailed
getCaUsageSetInfoFromElement	getCaUsageSetInfoFromElementSucceeded operationFailed
getCaUsageSetInfoFromConnector	getCaUsageSetInfoFromConnectorSucceeded operationFailed
getMultipleCaUsageSetInfo	getMultipleCaUsageSetInfoSucceeded operationFailed

InfrastructureSession Operation Descriptions

This section describes the operations defined in the WDIInfrastructure.IDL file.

Query Operation

- queryConditionCode

  Retrieves a list of condition codes that can be added to a mounting position or a port address. Criteria for the search is passed in the ConditionCode structure. See the ConditionCode structure in WDIInfrastructureTypes.idl for rules concerning the criteria. A WDITransaction object is intentionally not passed for this operation and assumes responsibility for transaction management.

Export Operations

- getStructureTypes

  Retrieves a list of all structure types in the database.

- getStructureFormatsGivenType

  Retrieves a list of active structure formats bound by the input structure type.

- getStructureFormatsGivenTypeAndArea

  Retrieves a list of active structure formats bound by the input structure type for a given input geographic area identifier.

- getComponentsGivenStructureFormat

  Retrieves a list of active structured format components based on the input active structured format.

- getValidValuesGivenStructureFormatComponent

  Retrieves a list of active valid values based on the input structured format component identifier.

- getGeoAreaTypes

  Retrieves a list of all geographic area types in the database.

- getGeoAreaTypesGivenCountry

  Retrieves a list of geographic area types used by the input country name.

- getGeoAreasGivenType

  Retrieves a list of the active geographic areas bound by the input geographic area type.

- getGeoAreasGivenTypeAndCountry

  Retrieves a list of the active geographic areas used by the input country name and bound by the input geographic area type.

- getRelatedGeoAreasGivenAreaAndType

  Retrieves a list of the active geographic areas related to the input geographic area and bound by the input geographic area type.

- getLanguages

  Retrieves a list of all languages in the database.

- getCodeCategories

  Retrieves a list of code categories bound by the input language code.

- getCodeCategoryValues

  Retrieves a list of code category values bound by the input code category number and language code. Only values with an effective from date on or before the current date, and values with a populated effective to date after the current date are returned. A null value is not returned.

- getConditionCode

  Retrieves the condition code, description, and warning type for a given condition code. The conditionCode parameter is required, and must contain a valid condition code, or an error is returned. This operation is intentionally not passed a WDITransaction object and assumes responsibility for transaction management.

- getNetworkAreasGivenLocation

  Retrieves network areas based on the location ID passed in.

- getNetworkAreasGivenLocation

  Retrieves network areas based on the location ID passed in.

- getCAUsageSetInfoFromServiceItem

  Retrieves CA usages based on the service item id passed in.

- getCAUsageSetInfoFromTemplate

  Retrieves CA usages based on the template id passed in.

- getCAUsageSetInfoFromElement

  Retrieves CA usages based on the element type passed in.

- getCAUsageSetInfoFromConnector

  Retrieves CA usages based on the connector id passed in.

- getMultipleCaUsageSetInfo

  Retrieves CA usages for multiple items passed in.

Export Customized Attribute Process Point IDs

The export customized attribute (CA) operations require you to enter a value for the process point ID. The following list describes the various process points in the application where CAs may be exported or rendered for display or update.

Table 4–3 lists the process point IDs.

Table 4–3 Process Point IDs

Export Customize Attribute Operation	Process Point ID	Description
Activation	60	Used during connection design change to request CAs for Connections.
Billing API	71	Used by the Billing API to request CAs.
Change Connection Design	60	Used during connection design change to request CAs for Connections.
Connection Design History	62	Used during connection design to display CAs for Connections history.
Disconnect Connection Design	61	Used during connection design for disconnect to request CAs for Connections.
GLR-Network Design Properties	58	Used by the GLR to show properties for a Connection.
Network Design - New	55	Used by Network Design to request CAs for a Network System or Network Element.
New Connection Design	59	Used during connection design to request CAs for New Connections.
Ordering - Change	51	Used by the Optimized Dialog to request CAs for a Change Order for an existing Network System, Network Element, or Connection.
Ordering - Disconnect	52	Used by the Optimized Dialog to request CAs for a Change Order for a disconnect of a Network System, Network Element, or Connection.
Ordering - New	50	Used by the Optimized Dialog to request CAs for a New Network System, New Network Element, or New Connection.

processPointId is an input for several Infrastructure export methods:

```
//getCaUsageSetInfoFromServiceItem- Retrieves ca usages based on the service item
//id passed in.
     void getCaUsageSetInfoFromServiceItem(
         in MetaSolv::CORBA::WDI::WDITransaction aWDITransaction,
         in WDINotification aWDINotification,
         in long serviceItemId,
         in long processPointId,
         in long referenceNumber)
         raises(MetaSolv::CORBA::WDI::WDIExcp);

//getCaUsageSetInfoFromTemplate- Retrieves ca usages based on the template
//id passed in.
     void getCaUsageSetInfoFromTemplate(
         in MetaSolv::CORBA::WDI::WDITransaction aWDITransaction,
         in WDINotification aWDINotification,
         in long templateId,
         in long processPointId,
         in long referenceNumber)
         raises(MetaSolv::CORBA::WDI::WDIExcp);
```

```
//getCaUsageSetInfoFromElement- Retrieves ca usages based on the element name
//passed in.
     void getCaUsageSetInfoFromElement(
         in MetaSolv::CORBA::WDI::WDITransaction aWDITransaction,
         in WDINotification aWDINotification,
         in string elementType,
         in long processPointId,
         in long referenceNumber)
         raises(MetaSolv::CORBA::WDI::WDIExcp);

//getCaUsageSetInfoFromConnector- Retrieves ca usages based on the connector id
//passed in.
     void getCaUsageSetInfoFromConnector(
         in MetaSolv::CORBA::WDI::WDITransaction aWDITransaction,
         in WDINotification aWDINotification,
         in long connectorId,
         in long processPointId,
         in long referenceNumber)
         raises(MetaSolv::CORBA::WDI::WDIExcp);

//getMultpleCaUsageSetInfo- Retrieves ca usages for multiple items passed in.
     void getMultipleCaUsageSetInfo(
         in MetaSolv::CORBA::WDI::WDITransaction aWDITransaction,
         in WDINotification aWDINotification,
         in MetaSolv::CORBA::WDIInfrastructureTypes::CaInputItemSeqitems,
         in long processPointId,
         in long referenceNumber)
         raises(MetaSolv::CORBA::WDI::WDIExcp);
```

NetworkLocationSubSession

The NetworkLocationSubSession is a subsession in the Infrastructure API that manages network location database transactions in the Infrastructure subsystem.

The NetworkLocationSubSession includes the following functionality described in detail in later sections:

- Associate and disassociate location relationships with a network location
- Associate and disassociate network areas from an end-user location
- Associate and disassociate tandem types from a network location
- Disassociate Secondary LSO's from an end-user location.
- Query, create, update and delete end-user locations
- Query, create, update and delete network locations
- Query network areas

The NetworkLocationSubSession interacts solely with the Infrastructure server which is responsible for *commit* and *rollback* functionality. The WDITransaction parameter is not a part of any operation used within this API. Prior operations that used the WDITransaction parameter are deprecated and replaced with new versioned operations.

NetworkLocationSubSession Interface Operations

Table 4–4 lists the operations in the NetworkLocationSubSession of the **WDINetworkLocation.IDL** file.

Table 4–4 Network Location Operations

Operation	WDINotifications
queryNetworkLocations This operation has been deprecated from the WDIEquipment.IDL and is replaced by queryNetworkLocations_V2	networkLocationQuerySucceeded - Deprecated networkLocationQueryFailed - Deprecated
getNetworkLocation This operation has been deprecated from the WDIEquipment.IDL and is replaced by getLocation	networkLocationGetSucceeded - Deprecated networkLocationGetFailed - Deprecated
queryNetworkLocations_V2	queryNetworkLocationsSucceeded_v2 (replaces networkLocationQuerySucceeded) operationFailed (replaces networkLocationQueryFailed)
queryEnduserLocations	queryEnduserLocationSucceeded operationFailed
queryNetworkAreas	queryNetworkAreaSucceeded operationFailed
getLocation	getLocationSucceeded (replaces networkLocationGetSucceeded) operationFailed (replaces networkLocationGetFailed)
createLocation	createLocationSucceeded operationFailed
updateLocation	updateLocationSucceeded operationFailed
deleteLocation	deleteLocationSucceeded operationFailed
getServingOfficeTypes	getServingOfficeTypesSucceeded operationFailed
getCentralOfficeExchangeAreas	getCentralOfficeExchangeAreasSucceeded operationFailed
getNetworkLocationCategories	getNetworkLocationCategoriesSucceeded operationFailed
getNetworkLocationTypes	getNetworkLocationTypesSucceeded operationFailed
getNetworkLocationRelationshipTypes	getNetworkLocationRelationshipTypesSucceeded operationFailed
getTandemTrafficCodes	getTandemTrafficCodesSucceeded operationFailed
getLocationCodeFormats	getLocationCodeFormatsSucceeded operationFailed
getTandemServices	getTandemServicesSucceeded operationFailed

Table 4–4 (Cont.) Network Location Operations

Operation	WDINotifications
getBuildingLocations	getBuildingLocationsSucceeded operationFailed
getAssociatedNetworkAreas	getAssociatedNetworkAreasSucceeded operationFailed
getAvailableNetworkAreas	getAvailableNetworkAreasSucceeded operationFailed
getTelephoneNumberSwitchLocations	getTelephoneNumberSwitchLocationsSucceeded operationFailed
getDataSwitchLocations	getDataSwitchLocationsSucceeded operationFailed
getTandemLocations	getTandemLocationsSucceeded operationFailed
getIncorporatedCodes	getIncorporatedCodesSucceeded operationFailed
getMultipleAddressPatterns	getMultipleAddressPatternsSucceeded operationFailed
associateLocationRelationships	associateLocationRelationshipsSucceeded operationFailed
unassociateLocationRelationships	unassociateLocationRelationshipsSucceeded operationFailed
associateTandemTypes	associateTandemTypesSucceeded operationFailed
unassociateTandemTypes	unassociateTandemTypesSucceeded operationFailed
associateNetworkAreas	associateNetworkAreasSucceeded operationFailed
unassociateNetworkAreas	unassociateNetworkAreasSucceeded operationFailed
unassociateSecondaryLSOs	unassociateSecondaryLSOsSucceeded operationFailed

NetworkLocationSubSession Operation Descriptions

This section describes the operations defined in the **WDINetworkLocation.IDL** file.

Query Operations

- queryNetwork Locations (Deprecated from the WDIEquipment.IDL)

  Requests and returns all network locations that match specific criteria.

- queryNetworkLocations_V2

  Requests and returns network locations for specific criteria, limiting the number of records returned.

- queryEnduserLocations

  Requests and returns end-user locations based on specific criteria.

- queryNetworkAreas

  Requests and returns network areas based on specific criteria.

Get Operations

- getNetworkLocation (Deprecated from the WDIEquipment.IDL)

  Retrieves a specific network location based on its network location ID.

- getLocation

  Retrieves an existing network or end-user location, or end-user location with network location alias.

- getServingOfficeTypes

  Retrieves servicing office types used with network location.

- getCentralOfficeExchangeAreas

  Retrieves central office exchange area values for use with network location.

- getNetworkLocationCategories

  Retrieves network location category values.

- getNetworkLocationTypes

  Retrieves types of network locations.

- getNetworkLocationsRelationshipTypes

  Retrieves types of network location relationships for use with network locations.

- getTandemTrafficCodes

  Retrieves tandem traffic code values used with network locations.

- getTandemServices

  Retrieves tandem service values used in network locations.

- getBuildingLocations

  Retrieves location codes for locations that are in buildings.

- getAssociatedNetworkAreas

  Retrieves network area associated with a particular end-user location.

- getAvailableNetworkAreas

  Retrieves available network areas that can be associated with a particular end-user location. This parameter is optional when creating an end-user location. This operation requires that a TN switch is associated with the end-user location.

- getTelephoneNumberSwitchLocations

  Retrieves location codes for telephone number switches used in end-user locations.

  The client application must specify a partial value that the API uses to filter values and return location codes meeting this criteria. The *telephoneNumberSwitchLocation* operation is optional when creating an end-user location. The user must specify at

least a partial value for the search. An end-user location can be created without a TN switch.

- getLocationCodeFormats

  Retrieves location code formats for network locations.

- getDataSwitchLocations

  Retrieves location codes for data switches used in end-user locations.

  The client application must specify a partial value that the API uses to filter values and return location codes for data switches meeting this criteria. The *getDataSwitchLocations* operation is operational when creating an end-user location. You must specify at least a partial value for the search. An end-user location can be created without the *getDataSwitchLocations* operation.

- getTandemLocations

  Retrieves location code values for tandem locations for network locations.

- getIncorporatedCodes

  Retrieves incorporated codes (inside incorporated area, outside incorporated area, and none) used to create a new end-user location.

- getMultipleAddressPatterns

  Retrieves patterns (odd, even, or both) for a range of end-user locations. The user provides the start and end range, and indicate if those within the range are odd numbered, even numbered, or both.

Create Operation

- createLocation

  Creates a new network, end-user location, or end-user location with network location alias and store it on the database.

Update Operation

- updateLocation

  Applies updates to a network, end-user location, or end-user location with network location alias in the database.

Delete Operation

- deleteLocation

  Removes a network or end-user location from the database. If the location is an end-user location with a network location alias, the user can choose to delete both the network and end-user location entries, delete just the end-user location entry, or delete the network location alias.

  Note: When network location type is B for an end-user location with a network location code alias, the user can delete just the alias, delete just the end-user, or delete both the end-user and network location entries.

Associate Operations

- associateLocationRelationships

Associates location relationships to a network location.

- associateTandemTypes

  Associates tandem types to a network location.

- associateNetworkAreas

  Associates network areas with a new or existing end-user location. Association of network areas to an end-user location can occur at the time the location is created. However, this class provides a more direct path for this association process so that existing end-user locations can have the association done without having to go through the update process. It assumes that a telephone number (TN) switch is specified for this end-user location. A TN switch must be set for the end-user location or an error will occur.

Unassociate Operations

- unassociateLocationRelationships

  Removes location relationships from a network location.

- unassociateTandemTypes

  Removes tandem types from a network location.

- unassociateNetworkAreas

  Removes network area associations from an end-user location. The switch network area will not be removed or disassociated through this process.

- unassociateSecondaryLSOs

  This operation unassociates secondary local servicing offices associated with an end-user location. Since end-user location is a type of network location, address and related formatting are identical to that of network location.

Process Flows

This section contains sample process flows for each type of signal: solicited and unsolicited. Use the sample flow as a template for developing your own process flows.

Solicited Messages

A solicited message is a message initiated by the MetaSolv Solution API servers. The Infrastructure API does not support solicited messages at this time.

Unsolicited Messages

An unsolicited message is a message initiated by the third-party software. The Infrastructure API plays the role of the server, and a third-party application plays the role of the client with the exception of the callback processing.

Sample Unsolicited Message Process Flow for Exporting Infrastructure Information

The overall process flow for exporting infrastructure information is as follows:

1. The third-party application binds to the MetaSolv Solution Application Server to get a WDIRoot object reference.
2. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.

3. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference and starts a database transaction.
4. The third-party application invokes the *startInfrastructureSession* operation of the WDIManager interface to get an InfrastructureSession object reference.
5. The third-party application instantiates a third-party implementation of a WDINotification object.
6. The third-party application invokes the desired operation of the InfrastructureSession object, passing the WDINotification object.
7. The Infrastructure server either returns the requested data structure asynchronously through invocation of the appropriate *Succeeded* operation of the WDINotification object, or returns exception information through invocation of the *operationFailed* operation of the WDINotification object.
8. The third-party application invokes the *destroyInfrastructureSession* operation of the WDIManager interface.
9. The third-party application invokes the *destroyTransaction* operation of the WDIManager interface.
10. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

5

The Inventory and Capacity Management API

The Inventory and Capacity Management (ICM) API provides the IDL for importing equipment and exporting circuits and equipment. The ICM API provides beginning-to-end visibility of service and network assets, including facilities, equipment, and circuits. By exposing equipment specifications and installed locations, as well as circuit, trunk, and facility capacity, the ICM API enables you to query for your capacity on facilities, trunks, PVCs, and SONET networks.

The ICM API also enables you to query for all equipment located at a network location, including all associated port information, hard-wired cross-connect information, and software cross-connect information.

The ICM API also enables you to take these actions in the Oracle Communications MetaSolv Solution database:

- Assign and unassign IP addresses
- Create, update, and delete network elements
- Create and destroy hard-wired cross-connects
- Create, update, and delete condition codes and comments for one or more physical port addresses or equipment mounting positions on a piece of equipment
- Install, update, move, copy, uninstall, and delete equipment
- Query for condition codes
- Query for IP addresses
- Query for network elements
- Query for network element types
- Validate network element type references

The CORBA-registered name for the API server process used by the ICM API is DLRSERVER.

> **Note:** The ICM API does not include the getSwitchActivation_V5 and getTransportProvisioning_V5 operations that are described in the IDL files used by the ICM API. The getSwitchActivation_V5 operations is enabled only if you have purchased a license for the Switch Provisioning Activation API. The getTransportProvisioning_V5 operation is enabled only if you have purchased a license for the Transport Provisioning Activation API.

Key MetaSolv Solution Concepts

This section of the chapter identifies and describes key concepts used in MetaSolv Solution that are also used by the ICM API.

Equipment Types, Equipment Specifications, and Equipment

In order to understand how MetaSolv Solution represents your equipment inventory in its database, you must understand the distinction between equipment types, equipment specifications, and individual pieces of equipment.

An equipment type is a broad categorization of the different kinds of equipment used in a telecommunications network, such as the types RELAY RACK, CHANNEL BANK, and CARD. All relay racks are categorized as type RELAY RACK, regardless of the manufacturer or part number. Details that differentiate one relay rack from another are defined in the equipment specifications for those pieces of equipment. Equipment type is a property of an equipment specification.

An equipment specification is a reusable definition of a specific kind of equipment. Equipment specifications identify the basic characteristics of a piece of equipment that are shared with other pieces of the same model of equipment, including:

- Equipment type
- Manufacturer
- Model number
- Number of physical mounting positions
- Number of logical port addresses
- Number of port address placeholders for each mounting position
- Transmission rates for each port address and port address placeholder

A piece of equipment is an instance of an equipment specification. An individual piece of equipment is a single, concrete piece of equipment that performs a function, such as a channel card, provides a service to other equipment, such as a jack panel, or houses other pieces of equipment, such as a relay rack. Information that is specific to a specific piece of equipment, such as serial number, is stored in the database record for that piece of equipment.

Typically, the smallest piece of equipment tracked in the MetaSolv Solution database is a card, such as a channel card. The individual electronic components that make up a card, such as buttons, fuses, transistors, capacitors, and diodes, are not normally included in the MetaSolv Solution equipment inventory. The physical connection ports on a piece of equipment are discussed later in this chapter.

In MetaSolv Solution, the cables, wires, and fiber strands that are also part of your network are not part of your equipment inventory. Instead, those items are part of your plant inventory. Plant inventory information and operations are available in the MetaSolv Solution Plant API. See "The Plant API" for more information.

Equipment Network Elements

Network elements represent intelligent devices that make up a telephony or data network and allow communication and transmission between different types of networks. A network element can be composed of a system with many shelves, such as a switch or digital cross-connect system (DCS), or it can be a SONET node. SONET nodes are defined in the MetaSolv Solution SONET network design module. Network elements are defined in the MetaSolv Solution Equipment Administration module.

Network elements can also be defined as gateway network elements (GNEs), allowing them to be communicated with locally, remotely, and through other network devices, such as a Network Management System (NMS). Defining a network element as a gateway network element enables you to log into that element, enabling communication and exchange transactions, such as software cross-connect commands. Defining remote access information is optional for network elements, but it is required for GNEs. GNEs must have one of the following fields or field combinations defined on the MetaSolv Solution Network Element Properties window - Node tab.

- The **IP Addr**, **Port**, and **Shelf** fields
- The **Dial Up** field
- The **Other** field

Target identifiers (TIDs) can be associated with multiple shelves through the network element, eliminating the need for separate identifiers at the shelf level when all the shelves are part of the same system, such as in the case of a switch or a DCS. TIDs are displayed on the CLR/DLR when an assignment is made to a card that is installed in a shelf that is associated with the element. TIDs are also displayed on the CLR/DLR when an assignment involves equipment that is associated with a node, whether it be through a physical assignment or an enabled port assignment. This also applies to network route assignments, network assignments, facility assignments, and equipment assignments.

Equipment Name Aliases

You can use an equipment name alias to give a second name to a piece of equipment. This enables you to refer to that piece of equipment by either name.

You might need to use equipment name aliases if a company you share data with uses a different naming convention than you do. For example, equipment or circuits that you do not own might be inventoried as part of your network. This might be required if equipment is located in a collocated environment, such as an associated Local Exchange Carrier's (LECs) or Inter-Exchange Carrier's (IXCs) building, where different names are used for equipment.

Equipment name aliases are displayed on the CLR/DLR in the Notes section, but can be suppressed when the associated design lines are suppressed.

Equipment Installation in MetaSolv Solution

Equipment installation is the process of selecting an equipment specification and associating it with a specific network location.

> **Note:** When importing equipment through the ICM API, you must import only one piece of equipment at a time. You cannot import an entire hierarchy of equipment with a single operation.

When you install a piece of equipment in MetaSolv Solution, you must:

- Indicate which equipment specification you want to use as the basis for the piece of equipment you are installing
- Specify additional details about the piece of equipment you are installing to distinguish it from other pieces of equipment installed at the same network location from the same equipment specification

If the equipment has defined mounting positions, you can install other pieces of equipment in those mounting positions.

In addition to installing equipment, you can:

- Move equipment between mounting positions at the same network location
- Move equipment between network locations
- Uninstall equipment (move equipment from the installed equipment hierarchy to the spare equipment hierarchy at a network location)
- Copy equipment definitions to additional mounting positions or to other locations
- Delete equipment from a network location
- Specify hard-wired cross-connects between port addresses on two pieces of equipment at the same network location
- Specify condition codes for any physical port addresses and mounting positions on an installed or spare piece of equipment
- Assign and unassign IP addresses to physical and virtual port addresses

Mounting Positions

A mounting position is a physical place on a piece of equipment where other equipment can be fastened or installed. For example, the mounting positions on a relay rack are a series of boltholes, while the mounting positions in a channel bank are a series of card slots. Other pieces of equipment can be fastened or installed in those mounting positions.

Note: The presence of a mounting position does not imply programmed or engineered capability to recognize, process or forward transmissions.

Mounting positions are only specified for equipment that has one or more places where other equipment is fastened or installed. For example, a D4 channel bank has 48 mounting positions. Therefore, the equipment specification for the D4 channel bank card indicates that it has 48 mounting positions. The channel cards, which occupy the D4 channel bank's 48 mounting positions, have no mounting positions. Therefore, the equipment specifications for the channel card indicate that they do not have mounting positions.

Ports and Port Addresses

Physical ports, also referred to as port addresses, usually provide the means to connect equipment in a network by using a plug and socket connection. A physical port is a physical location on a piece of equipment where signals enter or leave.

Because signals enter or exit, ports are assigned a rate code. The rate code assigned to a port implies the ability to attach a circuit with a rate code of equal value.

Virtual Port Addresses

Virtual ports are conceptual ports that do not physically exist on a piece of equipment. Virtual ports allow you to work with digital loop carrier (DLC) systems, where the capacity of the system is greater than the transport channels available. Virtual ports

also allow you to assign an IP address to a piece of equipment rather than to a specific physical port.

You can only assign circuits to the lowest level virtual ports. Once you assign a circuit to the lowest level (child) virtual port, the status of the parent-level virtual port remains unassigned and the status of the child-level virtual port changes as follows:

- If the circuit is associated with a service request, the circuit goes into "Pending" status immediately, and then into "In Service" status when the service request's Due Date task is completed.
- If the circuit is not associated with a service request, the circuit goes directly into "In Service" status.

Enabled Ports and Enabled Port Addresses

Unlike ordinary ports, an enabled port is not a physical place on a piece of equipment. Instead, it is a port that the equipment creates through its internal software. For example, a DCS is used to cross-connect channels from one facility to another. This connection is accomplished digitally through enabled ports. A DCS with two physical DS1 ports may have no mounting positions, but can still enable, through software, 24 ports for each DS1. The software-enabled ports are then used to cross-connect DS0 channels riding the DS1s.

The rate code for an enabled port address cannot exceed the rate code for the primary port address. The DCS in the example has two primary port addresses with DS1 transmission rates. Therefore, the DCS can enable only a DS1 or DS0 transmission rate port.

Port Address Placeholders

As a rule, mounting positions do not provide physical ports for attaching circuits. A port address placeholder is a construct in the MetaSolv Solution database that enables you to assign logical ports to mounting positions where equipment with physical ports is scheduled to be installed. In short, port address placeholders allow circuit design work to continue when equipment is not yet installed.

For example, you want to cross-connect a jack panel to a shelf before the shelf's cards are installed. However, at this point there are no port addresses to cross-connect to, because the port addresses are on the cards and the cards are not installed. The solution is to define placeholders for the shelf's mounting positions (the potential number of port addresses available once a card is installed in the mounting position). As a result, you can cross-connect to the port address placeholders before a card is installed. When you install a card, its port addresses are automatically associated with the mounting position's port address placeholders.

The act of installing equipment in a mounting position that has port address placeholders does associate the circuits as directed by the placeholders, but does not remove the underlying placeholders themselves. This enables you to move cards in and out of a mounting position without removing the underlying cross-connects. When the equipment is removed, the port address placeholder remains, awaiting the next equipment installed in that mounting position.

When you specify port address placeholders for a mounting position, verify that the number of port address placeholders match the number of ports on the equipment that is to be installed in those mounting positions. Also, the rate code assigned to the port address placeholders must match the rate code of the ports on the equipment you install.

Port Address Aliases

You can use a port address alias to give a second node address to a port. This enables you to refer to that port address by either node address.

You may need to use port address aliases if a company you share data with uses a different addressing format than the one you use. In a collocated environment, equipment or circuits that you do not own may be inventoried as part of your network.

Port aliases are included on the CLR/DLR in the Notes section, but can be suppressed when the associated design lines are suppressed. Notes in the Notes section of the CLR/DLR include port aliases to which circuits have been assigned or those that have been cross-connected to a port address. If the cross-connected port address has an alias, both aliases display.

Nodes and Node Addresses

A node is a piece of equipment on a network with the ability to recognize, process, or forward signals to other equipment. For example, a node can be a router in a token ring or an OC12 terminal in a SONET network.

A node is aware of other nodes on the network and is capable of receiving transmissions from or forwarding transmissions to other nodes. Like a letter delivered to its recipient through a series of postal centers, a communications signal travels across a network among nodes to reach its destination.

A node address is an identifier that is unique to each node, distinguishing one node from another. MetaSolv Solution can base node addresses on a hierarchy of the physical components comprising the node: rack, shelf, and card. You can manually replace or alter, override, this hierarchical (or concatenated) node address by using hard and soft node address overrides on individual ports. You can also define a sequential numbering scheme for mounting positions on an equipment specification in order to automatically number ports sequentially across cards in a shelf.

Sequential Port Address Numbering

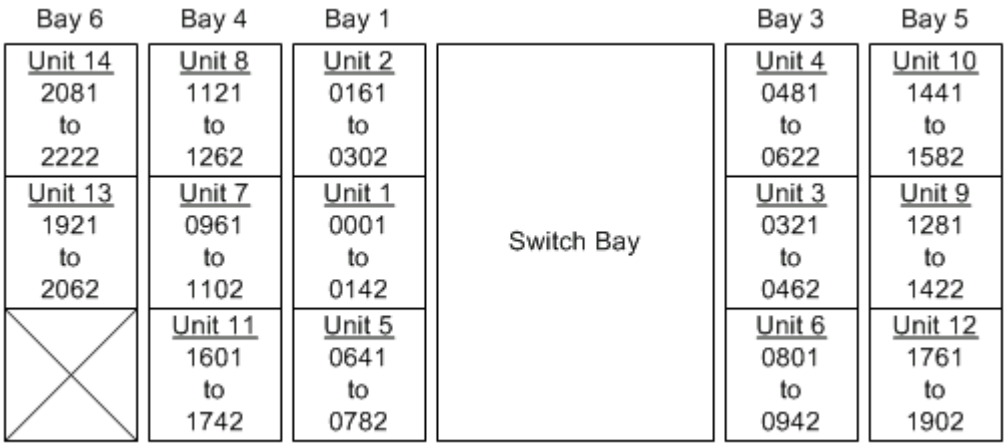
You can define a sequential numbering scheme for mounting positions on an equipment specification in order to number ports sequentially across cards in a shelf. You can use this automated numbering method, instead of hard and soft node address overrides, when working with multiple shelves of a DCS system where sequential numbering is applied to all ports of a given rate code. As with hard and soft node address overrides, the sequential numbering scheme you define replaces the concatenated node address.

Each shelf using sequential numbering is identified by a unique combination of unit number, unit extension, and network element location ID. The unit number identifies a piece or multiple pieces of equipment that contain cards. Every unit associated with a network element has a unique unit number and unit extension identifier. If the unit contains one shelf, that shelf has a unique unit number and a unit extension of zero.

For example, each of the 16 shelves in a Lucent DACSII Capacity Expansion Frame has a unique unit number between one and 16 and has a unit extension of zero. If the unit contains a group of shelves, each shelf in the group has the same unit number with a unique unit extension. Therefore, each of the four ATM shelves in a Lucent DACSII Single Bay has the same unit number with a unique unit extension. The first shelf is unit number one, unit extension one. The second shelf is unit number one, unit extension two, and so on.

The numbering sequence for card ports installed in a shelf's mounting positions is independent of the bay in which the unit is installed and independent of the order in which the units are installed. Thus, you can install Unit 5 in Bay 1 before you install Unit 4 in Bay 3 without affecting the numbering of the ports. Figure 5–1 illustrates sequential numbering of port addresses for DSPU cards in a Lucent DACSII.

Figure 5–1 Example of Sequential Numbering of Port Addresses



The sequential numbering scheme for a DCS shelf is defined on the equipment specification. You can create a variety of sequential numbering schemes, including straight sequential (with or without channel assignments) and sequential with augmentation. Upon installation of the shelf, you can disable numbering for selected ports by checking the Disable PA check box in the MetaSolv Solution Equipment window - Mounting Positions tab to create a sequential with skipped numbers scheme. Once the shelf is installed and a unit number and unit extension are defined, you cannot edit the sequential numbering scheme unless you uninstall the shelf. If a shelf is installed without a unit number and unit extension, and you add a numbering scheme to the equipment specification, the numbering scheme is not copied to the installed shelf unless you assign a unit number and unit extension and associate the shelf with a network element.

You can use the same specification to accommodate both sequential port numbering and hierarchical port numbering schemes. If you want to use a concatenated hierarchical port numbering scheme for a DCS systems, disable the numbering defined in the specification for each shelf in the DCS by unchecking the Seq Port Numbering check box on the MetaSolv Solution Equipment window - Mounting Positions tab.

Hard-Wired Cross-Connects

To a field engineer, a hard-wired cross-connect, also referred to as cabling, is the wiring of one equipment port to another. The hard-wired cross-connects you create in the MetaSolv Solution database represent the actual hard-wired cross-connects between equipment ports. An example of a hard-wired cross-connect is the cabling between a shelf and a DSX jack panel.

Hard-wired cross-connects remain intact as circuits are assigned or unassigned to cross-connected ports. In other words, an equipment port is dedicated to another equipment port so that when you assign a circuit to the first equipment port, through Circuit Design, the other equipment port is also included on the DLR/CLR for that

circuit. When the circuit is disconnected, the hard-wired cross-connect remains, awaiting the next circuit assignment.

You can create cross-connects in the MetaSolv Solution database to represent physical cross-connects that exist in your equipment inventory. You can make cross-connects between ports on a single piece of equipment or between ports on two separate pieces of equipment. You can also create cross-connects between a port address placeholder and a port address or port address placeholder. However, just as it is physically impossible to connect a given port address to itself, you cannot cross-connect port addresses and port address placeholders to themselves.

It is possible to cross-connect two pieces of equipment that have different Network Locations, allowing you to cross-connect equipment in two different locations. For example, in a collocated environment, you might want to cross-connect two pieces of equipment that are physically located in the same place but have different Network Location code assignments. When you use MetaSolv Solution to cross-connect equipment in two different locations, an informational message reminds the user that the locations are different.

The MetaSolv Solution cross-connect functionality enables you to create cross-connects for enabled port addresses. This functionality enables you to cross-connect equipment software to equipment hardware internally.

Several scenarios exist in which cross-connecting equipment is not allowed. Most of these scenarios relate to the existence of circuit assignments to one or both of the ports involved in the cross-connect.

The following scenarios describe instances when you cannot cross-connect equipment due to the existence of circuits that are already assigned to the port addresses being cross-connected.

- You cannot cross-connect a physical port address to an enabled port address if the physical port address has an assigned circuit and the enabled port address is already mapped.
- You cannot cross-connect a physical port address to an enabled port address if the physical port address has a circuit assignment and the enabled port address was not mapped.
- You cannot cross-connect a physical port address to another physical port address if both of the port addresses have different assigned circuits.
- You cannot cross-connect a physical port address that has a circuit assignment to a cross-connect chain that contains a mappable port.
- You cannot cross-connect an enabled port address to another enabled port address if both enabled port addresses are mapped to the same circuit and different circuit assignments exist for each enabled port address being cross-connected.
- You cannot cross-connect a physical port address to another physical port address if the port addresses have different pending assignments.

In the ICM API, an additional condition applies: if there is a “Blocked” condition code anywhere in the entire chain of circuits that would be created by a cross-connect, the ICM API does not create the requested cross-connect.

Condition Codes

Condition codes identify the condition of certain mounting positions, port addresses, or cable pairs. Using condition codes helps you prevent inventory from being used or better defines its capabilities. For example, if you wanted to mark a cable pair to no

longer be in service, you could give it a condition code of Bad. A **Local Assignment** condition code could denote that a port address has already been used on a local order. You can assign the type of warning that is given when an assignment is made to a circuit position, port address or cable pair with a certain condition code.

Circuit positions, mounting positions, and port addresses with condition codes are labeled **[Information]** or **[Blocked]** when you view them in the Equipment Install window or the Circuit Hierarchy window, depending on the condition code type.

IP Address Management in MetaSolv Solution

The MetaSolv Solution Infrastructure module includes an IP Address Management function that inventories all IP addresses owned by an ISP. IP addresses are unique numbers that identify a computer or device on a network. Public IP addresses are part of a standardized plan for identifying machines connected to the Internet. Using the IP Address Management function, you can keep track of IP addresses. The IP Address Management function lets you:

- Define base networks in your inventory
- Create subnets or pools from base networks
- Divide a subnet into more subnets
- View host IP addresses within a subnet
- Track the status of an IP address
- Query for existing IP addresses
- Combine subnets
- Create IP pools
- Delete subnets, IP pools, and base networks
- Recall IP addresses for reuse

The American Registry for Internet Numbers (ARIN) or your upstream ISP allocates base networks to you.

An IP address can be expressed as four decimal numbers separated by dots. Each number can have a value of zero to 255. An example of an Internet address is 130.5.5.171.

The size of base networks, which can be displayed as an IP address followed by a network prefix length (130.5.5.25/24). For example, a /24 network block has 256 IP addresses, where the first address is the subnet network address, the last address is the broadcast address and the remaining 254 addresses are host addresses. A network prefix can also be displayed as a subnet mask. For example, /24 is the same as a 255.255.255.0 submask.

In MetaSolv Solution, you can define your base network in one of two ways:

- You can divide your base network into two or more subnets of the same size.
- You can leave your base network as a pool of available addresses from which you can create subnets of varying sizes as you need them.

If you divide your base network into subnets and then divide any of the initial subnets into multiple smaller subnets, you can reverse this process by combining subnets to create a single larger subnet. You can delete a subnet if it is not assigned and none of its host addresses are assigned. When you delete a subnet, its unassigned addresses

become pooled addresses. Pooled addresses are not available for assignment. To be available for assignment, IP addresses must be part of a subnet.

When you define the base network, you can divide the IP address blocks into subnets or IP pools based on your business needs. Your specific business needs determine the number of subnets required and the size of each.

Overview of Assigning IP Addresses to Ports

You can assign an IP address to either a physical or virtual port. A physical port is a physical location on a piece of equipment where you can connect the equipment to a network by using a plug and socket connection. A virtual port is a conceptual port that does not physically exist on a piece of equipment. Virtual ports allow you to assign IP address generically to a piece of equipment, rather than to a specific physical port. You can assign to either a physical or a virtual port, depending on the situation. For example, if you are working with a router, you must assign an IP address to a specific serial (or physical) port. If you are working with a Web server, you assign the IP address for the customer's domain to the Web server and not to a specific port on that server.

The following rules apply to assigning IP addresses to port addresses:

- You can only assign one host number to a physical port.
- You can assign any number of subnets and/or host IP addresses to a virtual port address.
- You cannot assign IP addresses to a physical enabled port address or to a virtual enabled port address.
- If a subnet is assigned to a virtual port address, it must be at the lowest subnet level. The subnet cannot have any subnets defined below it.
- You cannot individually unassign host number IP addresses from a virtual port address once the subnet is assigned. You can only unassign the subnet.
- If you unassign the subnet from a virtual port address, all of the host numbers are also unassigned.
- A port address or its related equipment and an IP address may be associated with one or more network areas. The network area associated with a port address and its related equipment does not have to be the same as the network area associated with the IP address.
- Equipment connected by the same circuit must have IP addresses from the same subnet.
- You cannot assign the same circuit to more than two pieces of equipment with IP addresses.

An IP Address assigned to a physical port displays on the MetaSolv Solution Equipment Install window at the port address level. If a circuit is also assigned to the port address, the IP address displays before the Circuit ID. For example:

```
        (STS1 -In Service), 123.123.123.123, 1515 /ST01 /PLANTXXA
/PLANTXXB(In Service)
```

An IP Address assigned to a virtual port also displays on the MetaSolv Solution Equipment Install window at the port address level. Since multiple subnets and/or host numbers can be associated with the virtual port address, an IP address displays followed by the (…) symbol to indicate that more might exist. If a circuit is also

assigned to the port address, the IP address displays before the Circuit ID. For example:

```
        (STS1 -In Service), 123.123.123.123 (...), 1515 /ST01 /PLANTXXA
/PLANTXXB (In Service)
```

Some Common Questions About Equipment in MetaSolv Solution

This section identifies a number of questions MetaSolv Solution users commonly ask when first implementing the Equipment Administration module.

- How can I more quickly install equipment with the same configuration?

  If you have certain pieces of equipment that you install the same way repeatedly, create a "template" Network Location and copy the equipment from the template to real Network Locations.

- Can I inventory equipment that is stored in my warehouse?

  You can maintain a warehouse location that is used to inventory spare equipment. Make up a warehouse Network Location in which to "install" the equipment, then as the equipment is physically installed in its working location, move the equipment from the warehouse location to the working location.

- When I copy equipment, are associated condition codes also copied?

  No. When you copy equipment from one location to another, condition codes assigned to equipment positions are not copied.

- Should I define Slot Node and Port Addresses on equipment specs or during installation?

  Several scenarios exist that determine at what point you want to define slot node addresses and port addresses:

  – If the node address for an equipment type will always be the same, regardless of where it is installed, define the node address on the equipment specification.

  – If a single or common address exists for a specific piece of equipment, add the addresses to the shelf into which the equipment is installed.

  – If multiple ports exist on a card, and the address is always the same, add port addresses on the card's equipment specification.

  – If the node address for any type of equipment is determined when the equipment is installed in an office, add the slot node address or port address to each piece of equipment when it is installed.

See the online Help for more information.

ICM API Implementation Concepts

This section identifies key concepts you must know and key issues you must consider when developing applications that utilize the ICM API.

Transaction Management and the ICM API

The ICM API manages transaction processing on behalf of your application. That is, the ICM API handles all commits and rollbacks to the MetaSolv Solution database instead of requiring your application to explicitly commit or rollback transactions. When an operation you requested succeeds, the ICM API immediately commits the

results of the operation, then notifies you of the success of the operation. When an operation you requested fails, the ICM API immediately rolls back the results of the operation, then notifies you of the failure.

Note: Some of the older ICM API export operations still require you to supply a valid WDITransaction object reference. In these cases, you should still call the commit operation when using these export operations in order to release the read locks the database places on the exported records. Also in these cases, you must call the destroyTransaction operation in order to free the allocated resource.

Network Inventory Gateway Events and the ICM API

The ICM API and the MetaSolv Solution Network Management module support the use of gateway events for network inventory. Network inventory gateway events signal a third party that significant additions, deletions, or changes have occurred in the network inventory.

Network inventory gateway events are generated automatically based upon the settings of the rules/behaviors functionality in the MetaSolv Solution Work Management module. The actions in the Network Inventory module that can trigger evaluation of rules are:

- Installing or uninstalling equipment
- Copying equipment to a new location
- Modifying installed equipment
- Modifying condition codes for equipment mounting positions or port addresses individually
- Modifying condition codes for equipment mounting positions or port addresses by range
- Modifying virtual ports for equipment
- Moving equipment to an empty mounting position
- Assigning or unassigning an IP Address to equipment
- Assigning or unassigning a circuit to equipment
- Modifying hard-wired cross connects on equipment
- Modifying equipment specifications
- Modifying network elements on equipment

The actions listed above only trigger an equipment gateway event when the Work Management subsystem's rules and behaviors functionality is configured to do so.

Deleting equipment cannot trigger gateway events. When you delete installed equipment, the result of the deletion is that the equipment ID is removed from the MetaSolv Solution database. No equipment event can be sent in this case, because there is no equipment ID to pass.Therefore, you should uninstall the equipment to move it to "Spare" status, which can generate an equipment event, then delete the equipment.

DLR Mass Reconcile

When you edit equipment or equipment specifications, modify network elements, or move equipment with assigned circuits, the design layout reports (DLRs) for those circuits are reconciled to reflect the change, including pending assignments. The ICM API sends these reconciliations to the Background Processor utility. The ICM API does not support printing of design lines during DLR mass reconcile.

ICM API IDL files

The ICM API is described in these IDL files:

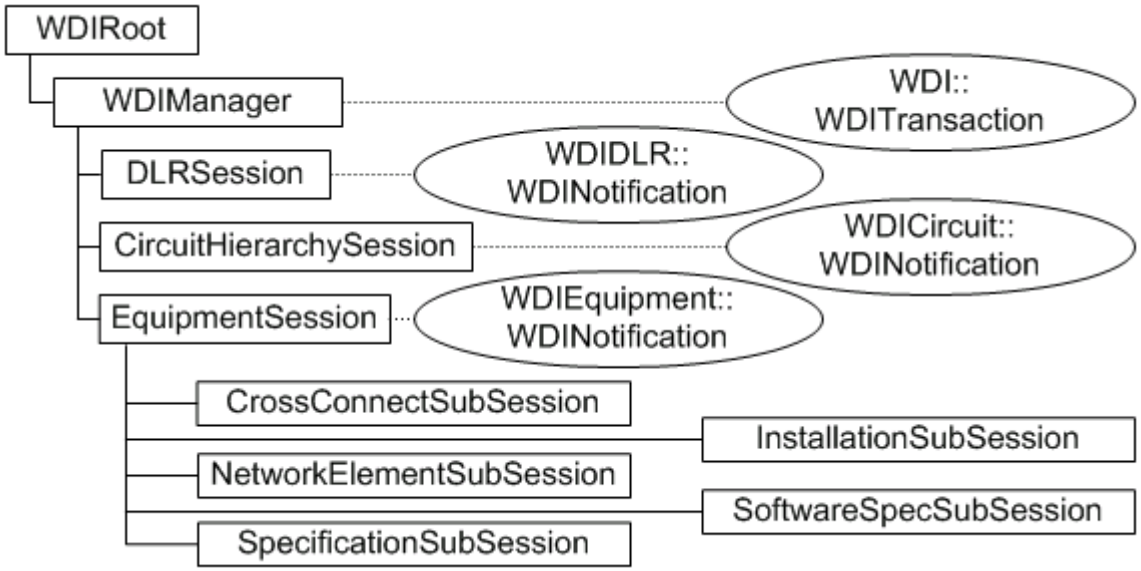
- WDI.idl
- WDICircuit.idl
- WDICircuitTypes.idl
- WDICircuitTypes_v2.idl
- WDICircuitTypes_v3.idl
- WDIDLR.IDL
- WDIDLRQueryTypes.idl
- WDIDLRQueryTypes_v2.idl
- WDIDLRQueryTypes_v3.idl
- WDIDLRTypes.idl
- WDIDLRTypes_v2.idl
- WDIDLRTypes_v3.idl
- WDIDLRTypes_v4.idl
- WDIDLRTypes_v5.idl
- WDIEquipment.idl
- WDIEquipmentTypes.idl
- WDIEquipmentTypes_v2.idl
- WDIEquipmentTypes_v3.idl
- WDIVLRTypes.idl
- WDIVLRTypes_v2.idl

The WDIPlant.idl and WDIPlantTypes.idl files are also included in the ICM API. See "The Plant API" for a complete description of the operations in these files.

ICM API Interfaces

Figure 5–2 shows the relationships of the modules and interfaces in the ICM API.

Figure 5–2 ICM API Interfaces



WDIManager Interface

Table 5–1 lists the operations available in the WDIManager interface of the **WDIDLR.idl** file.

Table 5–1 WDIManager Interface Operations

Operation	Description
destroyCircuitHierarchySession	Terminates the Circuit HierarchySession
destroyDLRSession	Terminates the DLRSession
destroyEquipmentSession	Terminates the EquipmentSession
destroyInSignal	Terminates the InSignal
destroyPlantSession	Terminates the PlantSession
destroySignal	Terminates the Signal
destroySignal2	Terminates the Signal2
destroyTransaction	Terminates the Transaction
startCircuitHierarchySession	Obtains the object reference for the Circuit HierarchySession
startDLRSession	Obtains the object reference for the DLRSession
startEquipmentSession	Obtains the object reference for the EquipmentSession
startInSignal	eventInProgress eventCompleted eventErrored
startPlantSession	Obtains the object reference for the PlantSession

Table 5–1 (Cont.) WDIManager Interface Operations

Operation	Description
startSignal	eventOccurred eventTerminated eventInProgress eventCompleted eventErrored
startSignal2	eventOccurred eventTerminated eventInProgress eventCompleted eventErrored
startTransaction	Obtains a handle to a database transaction

Note: See "Common Architecture" for a complete description of the operations.

CircuitHierarchySession Interface

Table 5–2 lists the operations available in the CircuitHierarchySession interface of the **WDICircuit.idl** file.

Table 5–2 CircuitHierarchySession WDINotification Operations

Operation	WDINotification
getBandwidthCircuits	getBandwidthCircuitsSucceeded getBandwidthCircuitsFailed
getCircuitPositionConditionCodes	getCircuitPositionConditionCodesSucceeded getCircuitPositionConditionCodesFailed
getCircuitPositionHierarchy	getCircuitPositionHierarchySucceeded getCircuitPositionHierarchyFailed
getCircuitPositionHierarchy_v3	getCircuitPositionHierarchySucceeded_v3 getCircuitPositionHierarchyFailed_v3
getCircuitPositionPending	getCircuitPositionPendingSucceeded getCircuitPositionPendingFailed
getCircuitPositionPrevious	getCircuitPositionPreviousSucceeded getCircuitPositionPreviousFailed
getNetworkRouteSegments_v2	getNetworkRouteSegmentsSucceeded_v2 getNetworkRouteSegmentsFailed
getNetworkSegmentCircuits_v2	getNetworkSegmentCircuitsSucceeded_v2 getNetworkSegmentCircuitsFailed_v2
getNetworkSegmentCircuits_v3	getNetworkSegmentCircuitsSucceeded_v3 getNetworkSegmentCircuitsFailed_v3

Table 5–2 (Cont.) CircuitHierarchySession WDINotification Operations

Operation	WDINotification
getTrunkGroup_v2	trunkGroupGetSucceeded_v2 trunkGroupGetFailed
queryNetworkRoutes_v2	queryNetworkRoutesSucceeded_v2 queryNetworkRoutesFailed_v2
queryTrunkGroups_v2	trunkGroupQuerySucceeded_v2 trunkGroupQueryFailed_v2
getTrunkGroupQueryValidValues_v2. This value is returned when implemented by the DLRSERVER.	N/A
getMaximumReturnedRows. This operation is implemented by the caller and returns a long. This allows the server to return maximum number of records for certain queries (0 = no limit.)	N/A

The following list contains a description of the operations available in the CircuitHierarchySession interface:

- getNetworkSegmentCircuits_v2

  Note: In MetaSolv Solution, when a SONET path-switched ring is built, two SONET routes are created. For example, for a 4 node ring, A-B-C-D, if the circuit requires entrance at node A and exit at node B, then there are two paths that can be traversed. These are A-B and A-D-C-B. Because of the rules surrounding a path-switched ring, MetaSolv Solution displays only one route, but combines the mileage and the connecting facilities under one segment tree item. This feature occurs on the display, even though both SONET routes are extracted from the database. However, the ICM API provides all of the SONET routes, with the ability to query each route individually. The assumption in the API is that, in the case of path-switched rings, the client program can combine the mileage and circuits for display purposes.

- getCircuitPositionHierarchy

  Note: Since hierarchy operations can return substantial amounts of data, a oneLevelOnly parameter is provided in the request structure to limit results to the first level of data directly beneath the request item for tree-structured data.

- getTrunkGroupQueryValidValues_v2

  If you pass empty criteria, the operation returns all valid values. If you pass match criteria for a field, the operation will return one QueryField full of matches for that field.

- getMaximumReturnedRows

Implemented by caller, returns a long value. This operations allows the API server to return the maximum number of records for certain queries (0 = no limit.)

EquipmentSession Interface Operations

Table 5–3 lists the operations available in the EquipmentSession interface of the **WDIEquipment.idl** file.

Table 5–3 EquipmentSession WDINotification Operations

Operation	Description
destroyCrossConnectSubSession	Terminates the CrossConnectSubSession.
destroyInstallationSubSession	Terminates the InstallationSubSession.
destroyNetworkElementSubSession	Terminates the NetworkElementSubSession.
destroySoftwareSpecSubSession	Terminates the SoftwareSpecSubSession.
destroySpecificationSubSession	Terminates the SpecificationSubSession.
startCrossConnectSubSession	Obtains the CrossConnectSubSession object reference.
startInstallationSubSession	Obtains the InstallSubSession object reference.
startNetworkElementSubSession	Obtains the NetworkElementSubSession object reference.
startSoftwareSpecSubSession	Obtains the SoftwareSpecSubSession object reference.
startSpecificationSubSession	Obtains the SpecificationSubSession object reference.

SpecificationSubSession Interface Operations

The SpecificationSubSession interface exposes operations for querying equipment specifications. Table 5–4 lists the operations and their corresponding WDINotification operations available in the SpecificationSubSession interface of the **WDIEquipment.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution.

Note: All failed operations in the SpecificationSubSession interface are reported through the generic operationFailed notification.

Table 5–4 SpecificationSubSession and WDINotification Operations

Operation	WDINotification
getEquipSpecQueryValidValues_v2. This value is returned when implemented by the DLRSERVER.	N/A
getEquipSpec_v2	getEquipSpecSucceeded_v2
getEquipType_v3	getEquipTypeSucceeded_v3
getUsageReport_v2	getUsageReportSucceeded_v2
queryEquipSpec_v2	queryEquipSpecSucceeded_v2
getEquipSpec_v3	getEquipSpecSucceeded_v3

SoftwareSpecSubSession interface operations

The SoftwareSpecSubSession interface exposes operations for querying software specifications. Table 5–5 lists the operations and their corresponding WDINotification operations available in the SoftwareSpecSubSession interface of the **WDIEquipment.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution. All failed operations in the SoftwareSpecSubSession interface are reported through the generic operationFailed notification.

Table 5–5 SoftwareSpecSubSession and WDINotification Operations

Operation	WDINotification
getSoftwareSpec	getSoftwareSpecSucceeded
querySoftwareSpec	querySoftwareSpecSucceeded

InstallationSubSession Interface Operations

The InstallationSubSession interface exposes operations for installing equipment and querying on installed equipment. Table 5–6 lists the operations and their corresponding WDINotification operations available in the InstallationSubSession interface of the **WDIEquipment.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution. All failed operations in the InstallationSubSession interface are reported through the generic operationFailed notification.

Table 5–6 InstallationSubSession and WDINotification Operations

Operation	WDINotification
addMountPosConditionCode	addMountPosConditionCodeSucceeded
addPortAddressConditionCode	addPortAddressConditionCodeSucceeded
assignIPAddress	assignIPAddressSucceeded
copyEquipment	copyEquipmentSucceeded
deleteEquipment	deleteEquipmentSucceeded
deleteMountPosConditionCode	deleteMountPosConditionCodeSucceeded
deletePortAddressConditionCode	deletePortAddressConditionCodeSucceeded
getEquipInstall_v2	getEquipInstallSucceeded_v2
getEquipInstall_v3	getEquipInstallSucceeded_v3
getEquipInstallMaint_v2	getEquipInstallMaintSucceeded_v2
getMountingPositionConditionCodes_v2	getMountingPositionConditionCodesSucceeded_v2
getPortAddressConditionCodes_v2	getPortAddressConditionCodesSucceeded_v2
getPortAddressInstall_v2	getPortAddressInstallSucceeded_v2
getPortAddressInstall_v3	getPortAddressInstallSucceeded_v3
getPortAddressIPAddress	getPortAddressIPAddressSucceeded
getPortAddressIPAddress_v2	getPortAddressIPAddressSucceeded_v2
installEquipment	installEquipmentSucceeded
moveEquipment	moveEquipmentSucceeded
queryEquipInstall_v2	queryEquipInstallSucceeded_v2

Table 5–6 (Cont.) InstallationSubSession and WDINotification Operations

Operation	WDINotification
searchEquipInstall_v2	searchEquipInstallSucceeded_v2
unassignIPAddress	unassignIPAddressSucceeded
uninstallEquipment	uninstallEquipmentSucceeded
updateEquipment	updateEquipmentSucceeded
updateMountPosConditionCode	updateMountPosConditionCodeSucceeded
updatePortAddressConditionCode	updatePortAddressConditionCodeSucceeded
validateNetworkElementMatch	validateNetworkElementMatchSucceeded
getEquipInstallQueryValidValues_v2. This operation is implemented by the DLRSERVER. It returns all valid values.	N/A

Comments Concerning Specific InstallationSubSession Operations

The ICM API does not support creation of equipment specifications. In order to install a piece of equipment, the equipment specification must already be defined in the MetaSolv Solution database.

Operations in the InstallationSubSession interface provide functionality equivalent to what exists in MetaSolv Solution to:

- Install a piece of equipment at a network location, including installing spare equipment
- Edit a piece of equipment at a network location
- Copy a piece of equipment, including:
  - Copying base equipment to a different location
  - Copying any equipment to or from spare
  - Copying non-base equipment to a different parent
  - Copying non-base equipment to different mounting positions within the same parent
- Move a piece of equipment, including:
  - Moving base equipment to a different location
  - Moving any equipment to or from spare
  - Moving non-base equipment to a different parent
  - Moving non-base equipment to different mounting positions within the same parent
- Uninstall a piece of equipment from a network location
- Delete a piece of equipment from a network location

You can use the InstallationSubSession interface operations listed below to perform the indicated functions:

- The queryEquipInstall_v2 operation queries first level equipment.
- The searchEquipInstall_v2 operation searches for a specific piece of installed equipment.

- The searchEquipInstall_v3 operation searches for a specific piece of installed equipment.
- The getEquipInstall_v2 operation returns the equipment tree for a piece of equipment.
- The getEquipInstall_v3 operation returns the equipment tree for a piece of equipment.
- The getPortAddressInstall_v2 operation returns port addresses for a piece of equipment.
- The getPortAddressInstall_v3 operation returns port addresses for a piece of equipment.
- The getEquipInstallMaint_v2 operation returns miscellaneous information for a piece of equipment.
- The getMountingPositionConditionCodes_v2 operation returns mounting position condition codes for a mounting position.
- The getPortAddressConditionCodes_v2 operation returns port address condition codes for a port address.
- The installEquipment operation installs a new piece of equipment.
- The updateEquipment operation updates information on an existing piece of equipment.
- The copyEquipment operation copies a piece of equipment to another location or mounting position.
- The moveEquipment operation moves a piece of equipment to another location or mounting position.
- The deleteEquipment operation deletes a piece of equipment.
- The uninstallEquipment operation move a piece of equipment to spare.
- The addMountPosConditionCode operation adds one or more condition codes to one or more mounting positions of a piece of equipment.
- The addPortAddressConditionCode operation adds one or more condition codes to one or more port addresses of a piece of equipment.
- The deleteMountPosConditionCode operation deletes one or more condition codes from one or more mounting positions of a piece of equipment.
- The deletePortAddressConditionCode operation deletes one or more condition codes from one or more port addresses of a piece of equipment.
- The updateMountPosConditionCode operation updates the comment for one or more condition codes on one or more mounting positions of a piece of equipment.
- The updatePortAddressConditionCode operation updates the comment for one or more condition codes for one or more port addresses of a piece of equipment.
- The validateNetworkElementMatch operation validates that the network element type associated to the input equipment specification is the same as the network element type associated to the input network element.
- The assignIPAddress operation assigns input IP addresses to the input equipment port address.
- The getPortAddressIPAddress operation retrieves IP addresses associated to the input equipment port address.

- The unassignIPAddress operation unassigns input IP addresses from the input equipment port address.

CrossConnectSubSession Interface Operations

The CrossConnectSubSession interface exposes operations for installing and querying on hardwired and software cross connects. Table 5–7 lists the operations and their corresponding WDINotification operations available in the CrossConnectSubSession interface of the **WDICircuit.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution. All failed operations in the CrossConnectSubSession interface are reported through the generic operationFailed notification.

Table 5–7 CrossConnectSubSession and WDINotification Operations

Operation	WDINotification
getHardwiredCrossConnects_v2	getHardwiredCrossConnectSucceeded_v2
getSoftwareCrossConnects_v2	getSoftwareCrossConnectSucceeded_v2
hwccRequest	hwccRequestSucceeded

For the most part, the CrossConnectSubSession interface operations duplicate the functionality of the MetaSolv Solution client. However, the API operations remove some of the restrictions the client imposes on making cross connects.

Any given piece of equipment can have four different type of port addresses:

- Port Addresses (PA)
- Enabled Port Addresses (EPA)
- Port Address Placeholders (PAPH)
- Virtual Enabled Port Addresses (VEPA)

In the CrossConnectSubSession, each of those four types of port addresses is considered a section. Hard-wired cross connections are made only for the ports belonging to a section at a time. The ICM API requires that all ports in the sequence be of the same type: PA, EPA, VEPA, or PAPH. Each section can repeat more than once, but intermingling of ports from different sections is not allowed. However, the FROM side port address type can be different from the TO side port address type.

For example, for two sets of starting port address numbers for cross connection on the FROM side, you specify [32, 20] and [102, 50]. For the corresponding TO side port addresses for connection, you specify [76, 20] and [210, 50]. The cross-connection process builds a FROM side list of 20 assignable ports for cross-connection starting from port 32, in ascending port order sequence, then builds a TO side list of 20 assignable ports starting from port number 76. Once both the FROM and TO lists are ready, the ICM API attempts the requested cross-connects.

Formats for Specifying FROM Side Port Addresses

FROM port addresses for hardwired cross-connects are specified in one of three formats. The FROM and TO side formats are independent, and any format on the FROM side can be combined with format case on the TO side.

- All ports format

The request is for cross-connecting all the ports on the FROM side equipment, starting from the first port on the FROM side. In this case the **PortAddrSeqFrom** has no entry.

- Specified range format

  The request is for cross-connecting a range of ports on the FROM side equipment. In this case, **PortAddrSeqFrom** has the range of ports for cross-connection. **portAddrSeqStart** contains the value of first port address of the range. **nbrOfPorts** specifies the number of ports to be cross-connected, starting from the port identified in **portAddrSeqStart**. For example, to cross-connect 100 ports from port number 132, specify [132,100]. To cross-connect an additional range of 50 ports starting from port number 760, follow the entry of [132,100] with a second entry of [760, 50].

  Note: In the example, if you specify 0 (zero) as the brOfPorts, that is, you specify [132,0], the operation connects all ports starting with port number 132.

  The range of specified ports cannot span across sections. To illustrate this using the preceding preceding example, assume that port number 132 is an Enabled Port Address (EPA). If port number 150 is NOT an EPA, the API gives a validation error. For all ports in a range to be cross-connected, all of the ports must be in the same section.

- Specified list of ports format

  The request is for cross-connecting a list of ports on the FROM side equipment. This situation can be treated as a special situation of the specified range format. In this case, the **PortAddrSeqFrom** has the list of ports for cross-connection. Each **portAddrSeqStart** contains the value of the port address to be cross-connected and the **nbrOfPorts** has a value of 1.

Formats for Specifying TO-Side Port Addresses

TO port addresses for hardwired cross-connects are specified in one of three formats. The FROM and TO side formats are independent, and any format on the FROM side can be combined with format case on the TO side.

- All ports format

  The request is for cross-connecting all the specified ports from the FROM side equipment to the TO side equipment, starting from the first port on the TO side. In this case, the **PortAddrSeqTo** has no entry.

- Specified range format

  The request is for cross-connecting all the specified ports from the FROM side equipment to the specified range of ports on the TO side equipment. In this case, the **PortAddrSeqTo** has the range of ports for cross-connection. Each **portAddrSeqStart** contains the value of the first port address of the range. The **nbrOfPorts** specifies the number of ports for cross-connection starting from the **portAddrSeqStart**. For example, to cross-connect 20 ports starting from port number 432, specify [432, 20]. To cross-connect an additional range of 75 ports starting from port number 320, the first entry of [432, 20] is followed by a second entry of [320, 75]. The range of specified ports can span across sections.

- Specified list of ports format

The request is for cross-connecting all the specified ports from the FROM side equipment to the specified list of ports on the TO side equipment. This can be treated as a special situation of the specified range format. In this case, the **PortAddrSeqTo** has a list of ports for cross-connection. Each **portAddrSeqStart** contains the value of the port address to be cross-connected to, and **nbrOfPorts** has a value of 1.

Comments concerning specific CrossConnectSubSession operations

Operations available in the CrossConnectSubSession interface:

- getHardwiredCrossConnects_v2

  Queries for hard-wired cross-connects.

- hwccRequest

  Requests creation of new hard-wired cross-connects.

- getSoftwareCrossConnects_v2

  Queries for software cross-connects.

NetworkElementSubSession Interface Operations

Table 5–8 lists the operations and their corresponding WDINotification operations available in the NetworkElementSubSession interface of the **WDIEquipment.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution. All failed operations in the NetworkElementSubSession interface are reported through the generic operationFailed notification.

Table 5–8 NetworkElementSubSession and WDINotification Operations

Operation	WDINotification
createNetworkElement	createNetworkElementSucceeded
createNetworkElement_v2	createNetworkElementSucceeded_v2
deleteNetworkElement	deleteNetworkElementSucceeded
getNetworkElement	getNetworkElementSucceeded
getNetworkElement_v2	getNetworkElementSucceeded_v2
getNetworkElementType	getNetworkElementTypeSucceeded
queryNetworkElement	queryNetworkElementSucceeded
queryNetworkElement_v2	queryNetworkElementSucceeded_v2
queryNetworkElementType	queryNetworkElementTypeSucceeded
updateNetworkElement	updateNetworkElementSucceeded
updateNetworkElement_v2	updateNetworkElementSucceeded_v2

Comments Concerning Specific NetworkElementSubSession Operations

You can use the InstallationSubSession interface operations listed below to perform the indicated functions:

- The createNetworkElement operation creates a network element.
- The createNetworkElement_v2 operation creates a network element.
- The deleteNetworkElement operation deletes a network element.

- The getNetworkElement operation retrieves the network element for the specified network node ID.
- The getNetworkElement_v2 operation retrieves the network element for the specified network node ID.
- The queryNetworkElement operation queries for a network element.
- The queryNetworkElement_v2 operation queries for a network element.
- The updateNetworkElement operation updates the specified network element.
- The updateNetworkElement_v2 operation updates the specified network element.

DLRSession Interface Operations

Table 5–9 lists the operations available in the DLRSession interface of the **WDIDLR.idl** file. These operations reproduce the same type of functionality as the corresponding function of MetaSolv Solution.

Table 5–9 DLRSession WDINotification Operations

Operation	WDINotification
getCircuitByWDIEvent	getCircuitByWDIEventSucceeded getCircuitByWDIEventFailed
getCircuitDLRs_v2	getDLRsByCircuitSucceeded_v2 getDLRsByCircuitFailed
getDLR_v2	DLRGetSucceeded_v2 (Deprecated) DLRGetFailed_v2 (Deprecated)
getDLR_v3	DLRGetSucceeded_v3 DLRGetFailed_v3
getDLR_v4	DLRGetSucceeded_v4 DLRGetFailed_v4
getDLR_v5	DLRGetSucceeded_v5 DLRGetFailed_v5
getDLRQueryOptionValues	This is a synchronous method so no notification method exists
getEndUserSpecialTrunkActivation_v2	endUserSpecialTrunkActivationGetSucceeded_v2 endUserSpecialTrunkActivationGetFailed
getEndUserSpecialTrunkActiviation_v4	endUserSpecialTrunkActivationGetSucceeded_v4 endUserSpecialTrunkActivationGetFailed_v4
getEndUserSpecialTrunkActiviation_v5	endUserSpecialTrunkActivationGetSucceeded_v5 endUserSpecialTrunkActivationGetFailed_v5
getEndUserSpecialTrunkTranslation_v2	endUserSpecialTrunkTranslationGetSucceeded_v2 endUserSpecialTrunkTranslationGetFailed_v2
getFlowThrough_v2	flowThroughGetSucceeded_v2 flowThroughGetFailed_v2
getQueryCircuits .	getQueryCircuitsSucceeded . getQueryCircuitsFailed

Table 5–9 (Cont.) DLRSession WDINotification Operations

Operation	WDINotification
getQueryCircuits_v2	getQueryCircuitsSucceeded_v2 getQueryCircuitsFailed_v2
getQueryDLRs_v2.	getDLRsByQuerySucceeded_v2. getDLRsByQueryFailed_v2.
getQueryDLRs_v3	getDLRsByQuerySucceeded_v3 getDLRsByQueryFailed_v3
getServiceRequestDLRs_v2	getDLRsByServiceRequestSucceeded_v2 getDLRsByServiceRequestFailed
getSwitchActivation_v2	switchActivationGetSucceeded_v2 switchActivationGetFailed_v2
getSwitchActivation_v4	switchActivationGetSucceeded_v4 switchActivationGetFailed_v4
getSwitchActivation_v5	switchActivationGetSucceeded_v5 switchActivationGetFailed_v5
getSwitchTranslation_v2	switchGetSucceeded_v2 switchGetFailed_v2
getTransportProvisioning_v2	transportProvisioningGetSucceeded_v2 transportProvisioningGetFailed
getTransportProvisioning_v4	transportProvisioningGetSucceeded_v4 transportProvisioningGetFailed_v4
getTransportProvisioning_v5	transportProvisioningGetSucceeded_v5 transportProvisioningGetFailed_v5
getVLR_v2	VLRGetSucceeded_v2 VLRGetFailed_v2
getDLRQueryOptionValues_v2. This operation is implemented by the DLRSERVER.	N/A
getMaskLocationCodes. This operation is implemented by the DLRSERVER.	N/A
getMaximumReturnedRows: This operation is implemented by the caller and returns a long. This allows the server to return maximum number of records for certain queries (0 = no limit.)	N/A

The ICM API does not have a generic query operation. However, you can use the getDLR_v5 query for most generic query purposes.

Process Flows

This section contains sample process flows for solicited and unsolicited messages. Use the sample flow as a template for developing your own process flows.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. MetaSolv Solution plays the role of the client and the third-party activation server plays the role of the server.

Sample Solicited Message Process Flow

When MetaSolv Solution is the client, the overall process flows as follows:

1. The client binds to the third-party server to get a WDIRoot object reference.
2. The client invokes the *connect* operation of the WDIRoot interface, and the *connect* operation yields a WDIManager object reference.
3. The client invokes the *startSignal* operation of the WDIManager interface to get a WDISignal object reference.
4. The client invokes the *eventOccurred* operation of the WDISignal interface to notify the third-party vendor that an event registered to them has occurred within MetaSolv Solution.
5. The client invokes the *destroySignal* operation of the WDIManager interface.
6. The client invokes the *disconnect* operation of the WDIRoot interface.

If the third-party application encounters an error, it throws a WDIExcp as defined by the IDL. The client handles CORBA system exceptions and WDIExcp exceptions.

Unsolicited Messages

An unsolicited message is a message initiated by the third-party application. MetaSolv Solution plays the role of the server, and a third-party application plays the role of the client with the exception of the callback processing.

Sample Unsolicited Message Process Flow for Exporting

The overall process flow for exporting a DLR follows:

1. The third-party application binds to the API server to get a WDIRoot object reference.
2. The third-party application invokes the *connect* operation of the WDIRoot interface, which then yields a WDIManager object reference.
3. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference and to start a database transaction.
4. The third-party application invokes the *startDLRSession* operation of the WDIManager interface to get a DLRSession object reference.
5. The third-party application instantiates a third-party implementation of a WDINotification object. The internal state of the client-supplied WDINotification object can be initialized, so the *getMaximumReturnedRows* function, when called by the server, returns the maximum number of entries to the client. If the function returns 0, entries for all objects matching the query criteria are returned.
6. The third-party application instantiates and populates a DLRQuery object. If necessary, the third party invokes the *getDLRQueryOptionValues* to obtain valid values to populate the DLRQuery object.

7. The third-party application then invokes the appropriate query operation on the DLRSession interface. In this example, DLRQuery, WDITransaction, and WDINotification are supplied as input parameters.
8. The API server invokes the operation DLRSession. The appropriate callback operation of the input WDINotification is called upon completion of the invocation of the DLRSession. In this example, the operations are *getDLRsByQuerySucceeded* and *getDLRsByQueryFailed*. The third party determines which circuit DLR to retrieve from the returned DLRResults.
9. The third-party application instantiates another WDINotification object and a DLRRequest structure, populated with the desired circuit and issue.
10. The third-party application invokes the *getDLR* operation of the DLRSession object, passing the *DLRRequest* and WDINotification object.
11. The DLR data structure is returned asynchronously through invocation of the *DLRGetSucceeded/Failed* operation of the WDINotification object.
12. The third-party application invokes the *destroyDLRSession* operation of the WDIManager interface.
13. The third-party application invokes the *destroyTransaction* operation on the WDIManager interface.
14. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

6

The Number Inventory API

The Number Inventory API was created to more efficiently handle the administration of telephone numbers and inventory items in Oracle Communications MetaSolv Solution. Operations are provided in the WDINI.IDL that provide the following functionality:

- Export Number Inventory
- Generate User ID
- Generate User Password
- Import Number Inventory
- Pre-assign Telephone Numbers
- Remove Inventory Association
- Update Number Inventory Provisioning
- Validate Password

The following operations provide lookup and export functionality:

- exportAccessTelephoneNumbers
- exportInventoryItem
- exportInventoryItemAssociation
- exportInventoryItems
- exportInventoryRelationTypes
- exportInventoryStatus
- exportInventorySubTypes
- exportInventoryTypes
- exportTelephoneNumbers
- exportTopLevelDomains

The following operations provide import functionality:

- importInventoryAssociation
- importNewInventoryItem
- importUpdatedInventoryItem

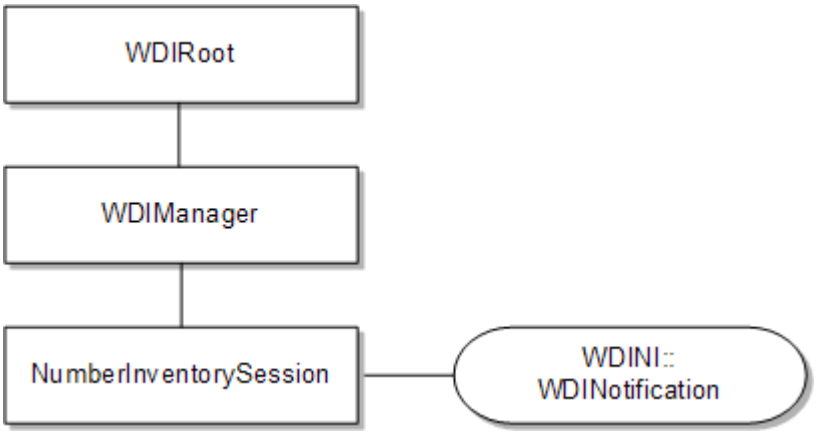
The WDINI.IDL file contains structures to support a flexible query. Fields on which you can specify search criteria include:

- Inventory Type Code
- Inventory Subtype Code
- Inventory Status Code
- Network Area City
- Network Area State
- Identify Text
- Identify Text Suffix.

Number Inventory API Interfaces

Figure 6–1 shows the relationship of the interfaces in the Number Inventory API.

Figure 6–1 Number Inventory API Session Interfaces



WDIManager Interface

Table 6–1 describes the operations in the WDIManager interface of the **WDINI.IDL** file.

Table 6–1 Number Inventory WDIManager Operations

Operation	Description
startNumberInventorySession	Obtains the object reference of the NumberInventory Session
destroyNumberInventorySession	Terminates the NumberInventorySession
startTransaction	commit rollback
destroyTransaction	Terminates the Transaction
startSignal	eventOccurred eventTerminated eventInProgress eventCompleted eventErrored
destroySignal	Terminates the Signal

Table 6–1 (Cont.) Number Inventory WDIManager Operations

Operation	Description
startInSignal	eventInProgress eventCompleted eventErrored
destroyInSignal	Terminates the Insignal

NumberInventorySession Interface Operations

Table 6–2 lists the operations and their notification operations in the NumberInventorySession.

Table 6–2 NumberInventorySession Interface Operations

Operation	WDINotification
exportNumberInventory	exportNumberInventorySucceeded exportNumberInventoryFailed
importNumberInventory	importNumberInventorySucceeded importNumberInventoryFailed
generateUserId	generateUserIdSucceeded generateUserIdFailed
generateUserPassword	generateUserPasswordSucceeded generateUserPasswordFailed
validatePassword	validatePasswordSucceeded validatePasswordFailed
updateNumberInventoryProvisioning	updateNumberInventoryProvisioningSucceeded updateNumberInventoryProvisioningFailed
exportTopLevelDomains	exportTopLevelDomainsSucceeded exportFailed
exportInventoryTypes	exportInventoryTypesSucceeded exportFailed
exportInventorySubTypes	exportInventorysubTypesSucceeded exportFailed
exportInventoryStatus	exportInventoryStatusSucceeded exportFailed
exportInventoryRelationTypes	exportInventoryRelationTypesSucceeded exportFailed
exportInventoryItem	exportInventoryItemSucceeded exportFailed
exportInventoryItems This operation uses the same succeeded operation as exportInventoryItem	exportInventoryItemSucceeded exportFailed
exportInventoryItemAssociation	exportInventoryRelationSucceeded exportFailed

Table 6–2 (Cont.) NumberInventorySession Interface Operations

Operation	WDINotification
importNewInventoryItem	importInventoryItemSucceeded importFailed
importUpdatedInventoryItem	importInventoryItemSucceeded importFailed
importInventoryAssociation	importInventoryAssociationSucceeded importInventoryAssociationFailed
removeInventoryAssociation	removeInventoryAssociationSucceeded removeInventoryAssociationFailed
exportTelephoneNumbers	exportTelephoneNumbersSucceeded exportTelephoneNumbersFailed * * The operation returns a failed notification and WDIError structure with an error when no data is found for a certain criteria.
preAssignTelephoneNumber	preAssignTelephoneNumberSucceeded preAssignTelephoneNumberFailed
exportAccessTelephoneNumbers	exportAccessTelephoneNumbersSucceeded exportAccessTelephoneNumbersFailed

Process Flow

The section that follows contains a sample process flow for unsolicited messages. Use the sample flow as a template when you develop your own process flows.

Unsolicited Messages

When the message is initiated by the third party (unsolicited), MetaSolv Solution plays the role of the server, and the third-party application plays the role of the client. Unsolicited messages are processed asynchronously, meaning a callback mechanism is used to report back the results of an operation invoked by the third-party application.

Sample Unsolicited Process Flow for Importing a Customer

The overall process flow for importing a customer is as follows:

1. The third-party application binds to the MetaSolv Solution Application Server to get a WDIRoot object reference.
2. The third-party application invokes the *startNumberInventorySession* operation of the WDIManager interface to get a NumberInventorySession object reference.
3. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.
4. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference.
5. The third-party application instantiates a WDINotification object.
6. The third-party application invokes the *importNewCustomer* operation on the NumberInventorySession interface, providing WDITransaction, WDINotification, and NumberInventory CustomerAccount objects.

7. The MetaSolv Solution Application Server processes the invoked operation of the NumberInventory Session and invokes the appropriate callback operation on the input WDINotification. In this example, the operations are *NumberInventoryExportSucceeded* or *NumberInventoryExportFailed* for exporting, and *NumberInventoryImportSucceeded* or *NumberInventoryImportFailed* for imports.
8. If the *NumberInventoryImportSucceeded* operation is invoked, the third-party application invokes the *commit* operation of the WDITransaction interface. If the *NumberInventoryExportFailed* operation is invoked, a WDIError sequence describing the error is returned to the third-party application. The third-party application then performs the appropriate error handling routine. In the case of an import failing, the third-party application should rollback the transaction.
9. The third-party application invokes the *destroyNumberInventorySession* operation of the WDIManager interface.
10. The third-party application invokes the *destroyTransaction* operation on the WDIManager interface.
11. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

Import Notifications

When the import of a new object succeeds, the document number is populated with the ID of the new record.

Number Inventory API Date Handling

To indicate that a date should be considered null, send 0 for the day, 0 for the month, and 0 for the year. If you supply a year that is less than four digits, 1900 is added to the value to determine the year. If four digits are provided, it is assumed that this is the exact year.

For example, if you provide 1/1/99, It is interpreted as January 1, 1999. If you provide 1/1/101, it is interpreted as January 1, 2001. If you provide 1/1/1, it is interpreted as January 1, 1901. If you provide 1/1/2001, it is interpreted as January 1, 2001.

7

The Activation API

The Activation API supports auto-activation for networks and connections. You must activate these networks or connections after designing, ordering, and provisioning them. Technologies include:

- ATM (Asynchronous Transfer Mode)
- DSL (Digital Subscriber Line)
- DSL with POTS (Plain Old Telephone Service)
- Ethernet
- Frame Relay
- MPLS (Multiple Protocol Label Switching)
- Traditional POTS
- VLAN
- VoATM
- VoDSL (Voice over DSL)

The export design provides the raw data that a third party's system needs to automatically activate a previously provisioned network or connection. The data includes information about system activation, activation of physical and virtual connections, and the elements which the connections link.

Connections

The export presents connection information by grouping connection and port address information under Network Elements. Connections represent the elements tied together because connections cannot exist without elements or ports. For the VLAN type of network, no connections exist like other systems, except for PortAddress assignments on the NetworkElements that make up the system. For this case, the port address assignments are shown on the NetworkElements separate from the individual and group connections on the element.

Network System Information

The NetworkSystem structure is returned in the NetworkSystems sequence on the Activation structure when a system is part of an order or the non-order specifies a network system. In the case of an order, if no elements or connections are ordered with the system, only the system information, including custom attributes, are returned. If elements or connections are ordered with it, they are returned with the system. In the

non-order scenario, all elements and connections related to the system are retrieved to show a complete view of the entire system.

Order Processing

Auto-activation provides the export of data necessary for activation and provides data for the Activation Report presented to a user online. The processing for the Network System Connection Export is predicated on gateway event processing. As part of the provisioning plan, an activation gateway event fires. This event includes information identifying activated networks or connections. In the case of processing by an order, the gateway event includes the WDIEvent, which has the order number as part of its data.

If this is a non-order activation, then the gateway event uses WDIEvent2 with the first key specifying the type of item and the second key specifying the item. The third key is used for the issue number where necessary.

The first key values are:

- 1 = virtual connections
- 2 = physical connections
- 3 = service items

The second key contains:

- For virtual connections, the design ID
- For physical connections, the circuit design ID
- For service items, the service item ID

The third key holds the issue number for the design in the case of virtuals and for the circuit design ID for physicals. When the first key indicates the request is for a physical connection, the third key representing the issue number is ignored because issue numbers are not applicable to such connections in this release.

Single Connection

Processing outside the context of an order can occur. A typical scenario involves a network already provisioned, active, and is reproduced within the MetaSolv Solution as inventory. If the calling application knows the network system ID or the connection in question, then export of the data for activation can occur. The calling application sends the WDIEvent2 structure passed to it by the gateway event and echoes it back to the API if:

- The type (first key on WDIEvent2) is virtual, the second key on the WDIEvent2 is the design ID of the virtual connection and the third key is the issue number.
- The type is physical, the second key on the WDIEvent2 is the circuit design ID of the physical connection and the third key is the issue number (ignored in this release).
- The type is service item, the second key is the service item ID and the third key is ignored.

Retrieval

In an order activation scenario, the order number passed in through the WDIEvent object drives the process. The document number is used to retrieve all level one service

items for the order and the types are evaluated. If the type is System, the child service items for that level one are retrieved and those types are evaluated. If the type is Connector or Element, the information for those items is retrieved. Connectors are represented by the elements which the connections bring together. All elements and any connectors grouped with those elements, associated with the network system, are returned. If the level one item is Element or Connector type, those Element structures are grouped on the Activation structure in the NonSystemSpecificElementsAndConnections sequence because connections are grouped under the elements they tie together. Switch Translation information is retrieved, if any exists for the order, and returned in the SwitchTranslations sequence on the Activation structure. Internet translation information, which appears on the Activation Report, is not included on the export.

MetaSolv Solution Key Concepts

In order to understand the information made available through the Activation API, you must understand certain key concepts. These concepts include:

- MetaSolv Solution Work Management subsystem
- MetaSolv Solution Gateway Event Server
- Gateway events and the Activation API
- Exporting data using the Activation API
- Reference architecture
- Design considerations

Activation API IDL files

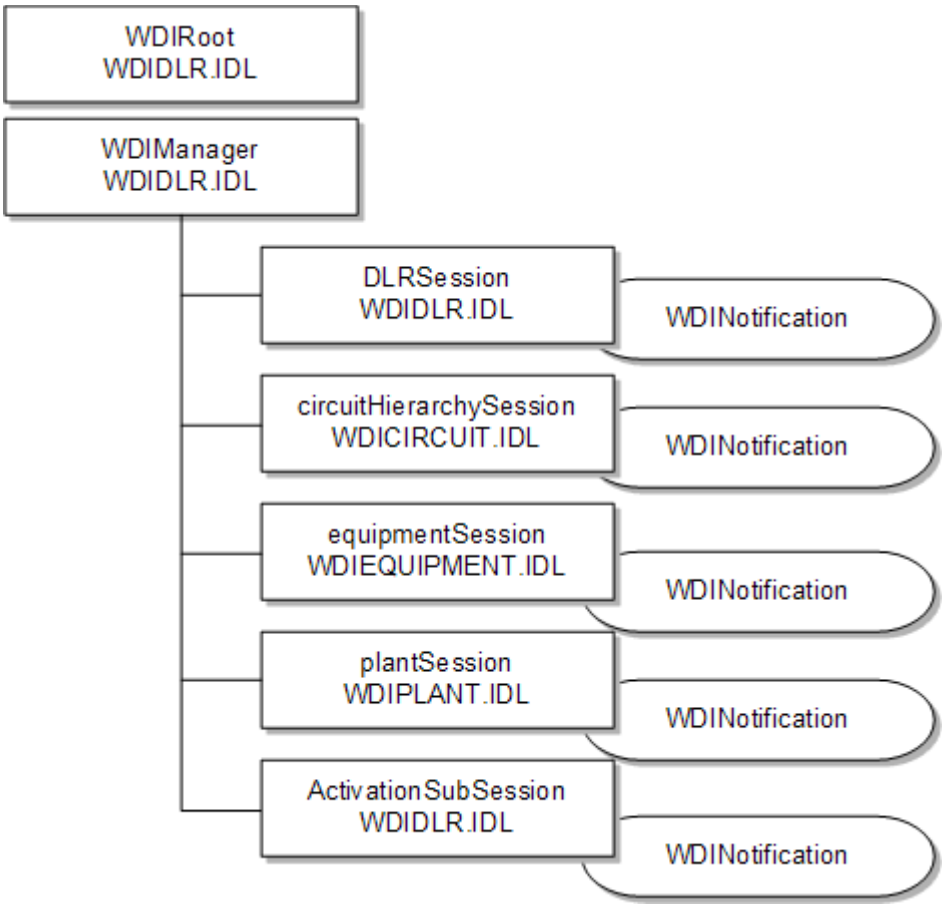
The Activation API consists of the following IDL files:

- WDIACTIVATIONTYPES.IDL
- WDI.IDL
- WDIUTIL.IDL
- WDIDLR.IDL

Activation API Interface Relationships

Figure 7–1 illustrates the relationship of the Activation API interfaces.

Figure 7–1 Activation API Interface Relationships



Activation API Operation Descriptions

The following IDL operations support the exporting of an activation assignment using the Activation API.

Only physical connections that have been completely designed are included in the Activation Report. Additionally, if the equipment used by a designed physical connection does not have an element association, the physical connection information for that element is not included on the Activation Report. The same is true for virtual connections with port addresses.

- getActivationInformationForOrder

  This operation drives the processing for the retrieval of the activation information using the data sent into the constructor. The operation performs the following tasks:

  – Retrieves the level one service item information for the document number. Evaluates the type for each of the service items. Processes items that have the type:

    * SYSTEM

    * PRDBUNDLE

  – Retrieves SwitchTranslation information for the document number if any is available.

  - Retrieves InternetTranslation information for the document number.
  - Retrieves all of the notes for the document number.
  - Retrieves the order information for the document number and populates the OrderInformationData using the information retrieved.
  - If the type is System, the operation retrieves the activation information for the network by performing the following tasks:
    * Obtains the Network System ID using the servItemId.
    * Retrieves the NetworkSystemData.
    * Retrieves the child service items for the system service item ID.
    * If child type is Connector, uses the circuit design ID from the service item record to retrieve a vector of NetworkElementData containers representing the connection.
    * If the child type is Element, retrieves the ID for the element. Retrieves the Network ElementData container for that element.
  - If the type is Product Bundle, the operation retrieves the activation information for the bundle by performing the following tasks:
    * Retrieves the child service items for the product bundle's service item ID and processes each one.
    * If the child type is Connector, uses the circuit design ID from the service item record to retrieve a vector of NetworkElementData containers representing the connection.
    * If the child type is Element, retrieves the ID for the element. Retrieves the NetworkElementData container for that element.

- getActivationInformationForVirtualConnection
  - Retrieves the NetworkElementData containers representing the virtual connection that includes allocated physical connections and allocated port addresses.
  - Adds NetworkElementData containers obtained to the NonSystemSpecificElementsAndConnections vector on the ActivationData container, which is translated before being sent back to the calling application.
- getActivationInformationForPhysicalConnection
  - Retrieves the NetworkElementData containers representing the physical connection.
  - Adds NetworkElementData containers obtained to the NonSystemSpecificElementsAndConnections vector on the ActivationData container, which will be translated before being sent back to the calling application.
- getActivationInformationForServItem
- Retrieves the service item record for the service item ID passed.
- If the service item is a physical connection, data retrieved is the same as calling *getActivationInformationForPhysicalConnection*.
- If the service item is a virtual connection, data retrieved is the same as calling *getActivationInformationForVirtualConnection*. The issue number used for the virtual

connection is the most recent pending issue, or if no pending issue exists, the most recent current issue.

- If the type is SYSTEM, obtain the network system ID using the service item ID.
  - Retrieves the NetworkSystemData for the network system ID.
  - Obtains the related virtual connection IDs and obtains the NetworkElementData containers representing those connections.
  - Obtains the physical connection IDs for any physical connections that do not have virtual allocations made to them. For each of these connections, obtains the NetworkElementData containers representing the appropriate element.
  - Obtains any remaining network element IDs. For each element ID, obtains the NetworkElement information by invoking the *getNetworkElementInformation.*
  - Adds the NetworkElementData containers for the elements and the connections. Adds them to the NetworkElements vector on the NetworkSystemData container.
  - Adds the NetworkSystemData container to the ActivationData container.
- If the type is Product Bundle, obtains the product bundle components by performing the following tasks:
  - Retrieves the child service items for the product bundle's service item ID and processes each one.
  - If the child type is Connector, uses the circuit design ID from the service item record to retrieve a vector of NetworkElementData containers representing the connection.
  - If the child type is Element, retrieves the ID for the element. Retrieves the NetworkElementData container for that element.
- If the type is ELEMENT, obtains the element ID.
  - Obtains the NetworkElementData container.
  - Adds the NetworkElementData container to the NonSystemSpecificElementsAndConnections on the ActivationData container.
  - Returns the ActivationData container to be translated and returned to the calling application.
- getActivationInformationForServItemWithOrderHeader

  Retrieves all of the data under the *getActivationInformationForServItem* heading. In addition:
  - Retrieves SwitchTranslation information for the document number if any is available.
  - Retrieves InternetTranslation information for the document number.
  - Retrieves all of the notes for the document number.
  - Retrieves the order information for the document number and populates the OrderInformationData using the information retrieved.
- getNetworkSystemInformation
  - Obtains the service item ID for the Network System ID.
  - Using the service item, invokes the *getActivationInformationForServItem* operation.

– Returns the NetworkSystemData container from the ActivationData container.

8

The Plant API

The telecommunications industry uses the term plant to describe two different environments within the context of network inventory management and network provisioning. These environments are outside plant (OSP) and inside plant (ISP). A company's inside plant investment is sometimes referred to as central office equipment (COE) or simply equipment.

The purpose of the Plant API is to enable the integration between an OSP system and Oracle Communications MetaSolv Solution. The primary intent is for the OSP to maintain plant inventory while the MetaSolv database retains assignment information.

The integration is achieved through gateway events, which are associated with tasks in a provisioning plan.

Plant implementation Concepts

This section describes issues you must be familiar with when building an application that interfaces with the Plant API.

Order Management

The MetaSolv Solution Work Management module assists MetaSolv Solution users in managing the flow of work and information from service requests to provisioning ordered services. Tasks are generated in the Order Management subsystem when the MetaSolv Solution user selects a provisioning plan upon completion of the order entry activities. A provisioning plan is a list of tasks required for each order type to be considered complete. Each task has a time interval and an assigned work group, responsible for completing the task.

The MetaSolv Solution Infrastructure module provides the MetaSolv Solution user with the ability to build and customize provisioning plans specific to their needs. The samples are primarily meant to reflect the sequential relationships between the PA, RID/DLRD, DD, and PAC tasks.

Associating the plant assignment gateway event with the PA task instead of the RID/DLRD task provides several advantages. The DLR/CLR lines do not show the correct plant assignments until the gateway event is complete. If the gateway event is associated with the RID/DLRD task, a MetaSolv Solution user opening the DLR/CLR prior to the completion of the plant assignment gateway event is presented with incomplete plant assignments. The user can avoid confusion if the gateway event is separated from the RID/DLRD task. Keeping the gateway event task (the PA task) separate from the RID/DLRD task enables smoother problem resolution if gateway event errors exist.

A provisioning plan sample for a new or change PSR order can include the following tasks:

- APP: to process the order application from customer
- CKTID: to identify circuit assignments and locations
- PA: to send gateway event for auto-assignment of plant
- RID: to complete the circuit design
- PTD: to perform plant test activities
- DD: to indicate the circuit is in service
- PAC: to send gateway event to indicate the plant is in service
- BILLING: to perform billing activities

A provisioning plan sample for an ASR or PSR disconnect order can include the following tasks:

- PA: to send gateway event for plant disassociation
- RID/DLRD: to disassociate plant from circuit and to complete other circuit disconnect activities
- DD: to indicate the circuit is disconnected
- PAC: to send gateway event to indicate plant is disconnected
- BILLING: to perform billing activities

A provisioning plan sample for an ISR may include the following tasks:

- PA: to send gateway event for auto-assignment of plant
- RID: to complete the circuit design
- PTD: to perform plant test activities
- DD: to indicate the circuit is in service
- PAC: to send gateway event to indicate the plant is in service

Recommendations for Assigning Gateway Events to Provisioning Plan Tasks

Before you can associate gateway events with a provisioning plan task, the MetaSolv Solution user must first define the gateway and gateway event in the MetaSolv Solution Work Management Gateway module. While developing a provisioning plan, the MetaSolv Solution user can associate gateway events to specific provisioning plan tasks.

> **Note:** When creating a gateway event, the MetaSolv Solution user must negotiate with the mediation server vendor to define appropriate gateway event names and platform-related information.

You cannot associate gateway events with orders that already have provisioning plans applied. Therefore, you should add gateway events to any task that might be used for an electronic interface in the future. After task generation, a MetaSolv Solution user can bypass a gateway event or reactivate a bypassed gateway event for a task that has not completed. The MetaSolv Solution user can also reopen a task, and then reactivate the gateway event for completed tasks.

You can assign multiple gateway events to a single task. You can also assign a gateway event to multiple tasks. When you assign a gateway event to a task, the task cannot be completed until the gateway event is complete.

Provisioning plan tasks can be defined as system tasks. System tasks do not require any action by the MetaSolv Solution user.

The plant assignment and inventory interface can be accomplished with only two different gateway events: Plant assignment and plant assignment complete. According to the provisioning plan samples illustrated above, the plant assignment gateway event is associated with the PA task, and the plant assignment complete gateway event is associated with the PAC task.

When defining these events, the following parameters are recommended. You should check the **Force Reopen** check box so that the gateway event can be resubmitted in the event that a task is reopened due to a supplement to the order prior to completing the order. For the event level, select **Order Level** so that a single gateway event signal is sent for all of the circuits requiring plant assignment. Specify the Direction as outbound. You should associate the gateway events with all three of the activity groups: new, change and disconnect. Check the **Provisioning** check box for the event type.

Options for Modify Cable Pair Assignment Preference

The Plant API requires you to set the value for the system preference Options for Modify Cable Pair Assignment to Create Pending Assignment. The four options presented in this preference dictate how MetaSolv Solution is to manage the assignment if the requested plant element is already assigned to another circuit. The Plant API assumes a tight integration with the OSP system. It assumes that assignment statuses are synchronized between the OSP system and MetaSolv Solution. Therefore, the Plant API always attempts to create pending assignments when plant elements are reassigned for future-use circuits. Plant API and Plant Administration software options are mutually exclusive.

Transaction Management and the Plant API

The Plant API manages transaction processing on behalf of your application. That is, the Plant API handles all commits and rollbacks to the MetaSolv Solution database instead of requiring your application to explicitly commit or rollback transactions. When you request an operation that succeeds, the Plant API immediately commits the results of the operation, then notifies you of its success. When a requested operation fails, the Plant API immediately rolls back the results of the operation, then notifies you of the failure. The Plant API's importPlantAssignment operation, which allows processing of multiple circuits, performs the commit or rollback separately for each circuit as the import succeeds or fails, prior to notifying you of the result. If it fails, the importPlantAssignmentFailed notification returns the list of circuits that were successfully updated prior to the failure.

Associating Separations Route to Plant Transport

Plant API does not allow for the association of a separations route to the plant transport (cable complement). In MetaSolv Solution, you can specify a separations route for a given complement. The application uses the mileage in the separations route to validate the length of the plant element properties.

Consequential Equipment Assignments

The Plant API does not offer the ability to import equipment to assign along with the plant element assignments. Typically, a plant element terminates at the CO by a piece of line equipment or a fiber distribution panel. When the OSP sends the plant assignment information, it might know the line equipment on which the plant element terminates. The Plant API does not offer the capability to import, assign, and build DLR blocks for line terminating equipment. You can create the hard-wired cross-connect between the equipment you want to use for the assignment and the line-terminating equipment and manually assign the line equipment. Otherwise, the process of selecting and assigning the line-terminating equipment is manual.

Key MetaSolv Solution Concepts

In order to understand the information made available through the Plant API, you must understand certain key concepts. These concepts include:

- MetaSolv Solution Work Management subsystem
- MetaSolv Solution Gateway Event Server
- MetaSolv Solution Infrastructure API Server
- Gateway events and the Plant API
- Exporting data through the Plant API
- Importing data through the Plant API
- Reference architecture
- Design considerations
- Transaction management
- Structured formats

Plant API IDL Files

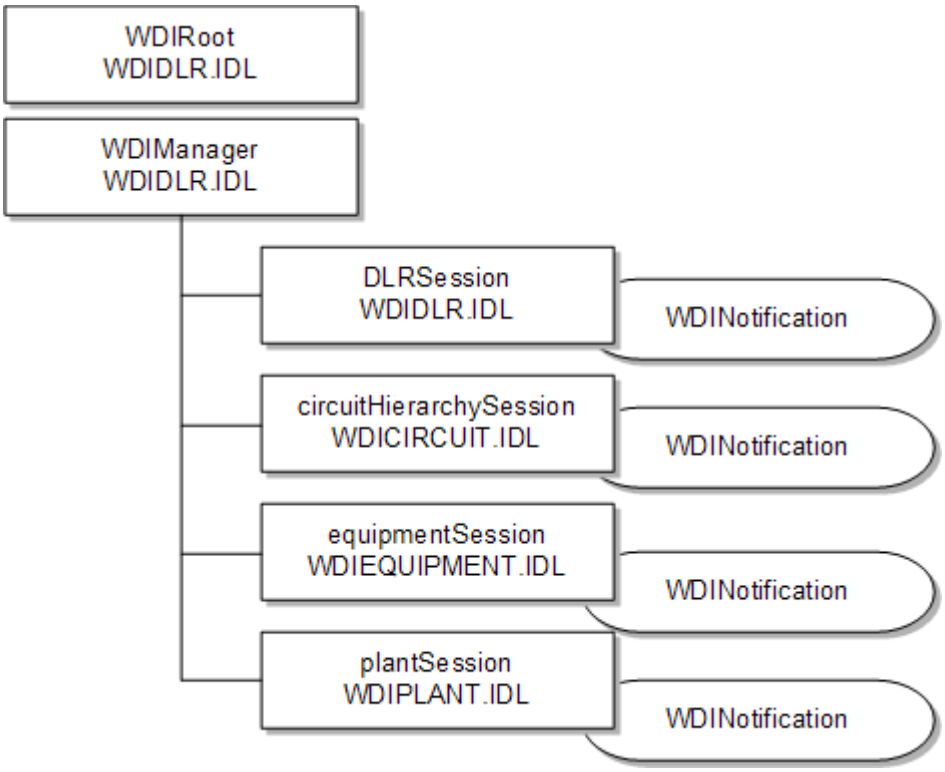
The Plant API consists of the following IDL files:

- WDIPlant.idl
- WDIPlantTypes.idl
- WDI.idl
- WDIUtil.idl

Plant API Interface Relationships

Figure 8–1 illustrates the relationship of the Plant API interfaces.

Figure 8–1 Plant API Interface Relationships



PlantSession Interface

Table 8–1 lists the operations that comprise the PlantSession interface.

Table 8–1 Plant API Interface Operations

Operation	WDINotification Operations
getFunctionCodes	getFunctionCodesSucceeded operationFailed
getLoadingTypes	getLoadingTypesSucceeded operationFailed
getPlantTransportClasses	getPlantTransportClassesSucceeded operationFailed
queryPlantTransportPhysical CompositionSpec	queryPlantTransportPhysicalCompositionSpec Succeeded operationFailed
getPlantElementAssignment Statuses	getPlantElementAssignmentStatusesSucceeded operationFailed
exportServiceRequestDetail	exportServiceRequestDetailSucceeded operationFailed
exportPlantAssignment	exportPlantAssignmentSucceeded operationFailed

Table 8–1 (Cont.) Plant API Interface Operations

Operation	WDINotification Operations
importPlantAssignment	importPlantAssignmentSucceeded importPlantAssignmentFailed

Plant API Operation Descriptions

The following IDL operations support the importing of a plant assignment using the Plant API.

- getFunctionCodes

  Retrieves a list of valid function codes for use with a plant assignment. The referenceNumber argument is generated by the operation's client, allowing it to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment.

  Function codes represent the uses for a plant element assigned to a circuit. You get the list of valid function codes from a static list. The list is as follows:

  - T = Transmit
  - R = Receive
  - S1 = Side One
  - S2 = Side Two
  - X2 = Two Wire
  - X4 = Four Wire

- getLoadingTypes

  Obtains a list of valid loading types to use to query for a plant transport physical composition spec. The referenceNumber argument is the number that generates by the client of the operation, allowing the client to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment.

  Loading types represent the possible ways in which a plant element amplifies to counteract signal loss. This list is user-definable and dynamic. The data provided as base data to a new customer is as follows:

  - D66: D indicates 4500 feet between load points with 66mh (millihenry).
  - H88: H indicates 6000 feet between load points with 88mh (millihenry).
  - NL: Non-loaded refers to cable pairs without load coils attached to them.
  - L: Loaded refers to cable pairs with load coils attached to them.

- getPlantTransportClasses

  Obtains a list of valid classes to use to query for a plant transport physical composition specification. The referenceNumber argument is the number that generates by the client of the operation, allowing the client to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment.

Plant transport classes represent the medium of the plant transport. This static list has the following values:

- COPPER
- FIBER
- MICROWAVE
- SATELLITE
- COAX

- queryPlantTransportPhysicalCompositionSpec

  A success notification operation that corresponds to the *queryPlantTransportPhysicalCompositionSpec* operation. The referenceNumber argument is the number that generates by the client of the operation which allows the client to match the asynchronous request to the corresponding result. The *plantTransportPhysicalCompositionSpecList* is the list of plant transport physical composition specs returned based on the query criteria you provide to the *queryPlantTransportPhysicalCompositionSpec* operation.

  The plant transport physical composition specifications describe the possible gauge, loading type, medium, and frequency combinations you can use to describe the physical aspects of a plant transport. This query equates to the Cable Pair Properties Query you can find in the Infrastructure module. A new generic query name now exists so that you can represent all mediums of plant transports (cables) without a bias toward one specific medium.

- *getPlantElementAssignmentStatuses*

  A success notification operation that corresponds to the PlantSession::*getPlantElementAssignmentStatuses* operation. The referenceNumber argument is the number that generates by the client of the operation, allowing the client to match the asynchronous request to the corresponding result. The assignmentStatusList argument is the list of valid plant element assignment statuses.

  Plant element statuses represent the possible list of statuses that a plant element (pair) can have. The possible values from this static list are:

  - 1 = Unassigned
  - 2 = Pending
  - 3 = InService
  - 4 = Pending Discount
  - 6 = Reserved
  - 7 = Reserved Capacity

- exportServiceRequestDetail

  Obtains a service request detail to determine how to process the gateway event signal. The third-party server receives a generic notification from the gateway event server and needs to determine the service request activity in order to decide whether to obtain new connection information from the OSP system to pass to the MetaSolv Solution Plant API or to disconnect the existing connection on the OSP system. Additionally, this operation obtains the absolute originating and terminating locations for a circuit on an order, not only the end point but also the local serving offices, if appropriate. Given these two endpoints, the client must use

the Infrastructure API to gather the location details for the location identifiers supplied.

Using these location details, the client must match the location with the same location in the integrated third-party database. Once this work is complete, the client must find the associated plant and location details to pass to the MetaSolv Solution Plant API. If the client requires additional locations to complete the connection, the client must once again use the MetaSolv Solution Infrastructure API to find query for the internal MetaSolv Solution location identifier which passes to the Plant API along with the other plant assignment information. The referenceNumber argument is the number that generates by the client of the operation which allows the client to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment. The documentNumber argument is the key to the service request.

You can obtain the document number from the MetaSolv::WDI::WDIEvent.documentNumber member. The WDIEvent is sent as a data payload from the gateway event server to the third-party server. If you are using this operation to obtain the list of circuits associated with an order, and the request to do so did not generate by the gateway event server, you must find the document number in the MetaSolv Solution database by matching service request details from the OSP to service request details in the MetaSolv Solution database, with the query yielding a MetaSolv Solution document number for the service request. You can either query using an existing API, such as PSR orders, or through a direct database SQL call.

- *exportPlantAssignment*

  Obtains the plant element assignment information and plant transports used for designing a given circuit. The referenceNumber argument is the number that generates by the client of the operation, allowing the client to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment. The circuitDesignIdentifiers argument references the internal circuit design ID for MetaSolv Solution. You must retain this information on the OSP system for reconciliation purposes. The third-party server can pass more than one circuit design identifier in order to retrieve the plant transport assignment data for multiple circuits.

- importPlantAssignment

  Imports the plant assignment data that passes from the client (the third-party mediation server). The concept is that an OSP system, other than MetaSolv Solution, maintains plant inventory. The OSP decides the appropriate plant data to use to complete the physical connection between the originating and terminating locations for the circuit, and provides that information to the MetaSolv Solution Plant API by means of the *importPlantAssignment* operation. The referenceNumber argument is the number that is generated by the client of the operation, allowing the client to match the asynchronous request to the corresponding result. The notification argument is the callback reference necessary for the API to complete the request in an asynchronous environment. The documentNumber argument is the key to the service request. You can obtain the document number from the MetaSolv::WDI::WDIEvent.documentNumber member or as output from the *exportServiceRequestDetail* IDL operation.

  The circuitDesignID argument represents one of the provisionable circuits that appear on the service request represented by the document number argument. The list of provisionable circuits can be determined using the *exportServiceRequestDetail*

operation. The circuitPlantTransportList argument is the list of plant transport details necessary to complete the physical connection between the two endpoints of the circuit. The list can contain the connection details for an unlimited number of circuits.

Note: The Plant API commits each circuit individually to avoid any problems with filling rollback segments on the MetaSolv Solution database.

In the event of an error (*importPlantAssignmentFailed* is called), the Plant API notifies the third-party server with the failed circuit design identifier.

Using that information, the third-party server can determine the success and failure of all of the circuits. Also, the *importPlantAssignmentFailed* notification returns a list of all the circuits designed successfully. This information can be useful for resetting statuses on the OSP system. If the third-party server does not want to handle the difficulties inherent in working with large sets of data, the server can choose to call the IDL operation, once for each circuit found on the service request.

Note: The Plant API does not offer the ability to restart processing where it failed when the gateway event restarts. Because Plant API does not have a way of knowing if the OSP wants to change its allocation, the mediation server can resend all the circuits with the original assignment information or send only those that had not processed at the point of failure.

MetaSolv Solution API Software and Mediation Server Processes

An essential advantage of the MetaSolv Solution API architecture is the integration between the OSS Gateways and the MetaSolv Solution Work Management subsystem. This integration is enabled by gateway events. Gateway events are inbound or outbound signals between the Work Management subsystem and a third-party gateway vendor. As tasks are started or completed, gateway event signals are initiated to notify the third party vendor. Once notified, the third party software application can take appropriate action based on the event.

The mediation server is responsible for implementing the MetaSolv Solution API operations. Upon receipt of a gateway event signal, the mediation server takes appropriate action. To support the MetaSolv Solution Plant API interface, the mediation server is responsible for responding to two different gateway events: plant assignment and plant assignment complete.

The following process flows illustrate sample interactions between the MetaSolv Solution gateway event server, the Plant API server, the Infrastructure API server, the third-party mediation server, and to some extent, the external Plant inventory application. The integrator is ultimately responsible for designing, developing and implementing the interface. The process flows are intended to present important concepts, which should be considered by the integrator when developing the interface. The flows are not intended to dictate how to implement the interface.

Request for Plant Assignment

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed an order for a new circuit. The MetaSolv Solution Work Management module places the PA task into a Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.
5. The activity code returned from the *exportServiceRequestDetail* operation is equal to New when a new circuit is ordered and plant facilities must be assigned to fulfill the order.
6. The third-party mediation server performs a process to obtain network location details, if necessary. See "Obtain Network Location Details" for more information. The *exportServiceRequestDetailSucceeded* operation provides the internal MetaSolv Solution location IDs for the absolute endpoints of a circuit. The third-party mediation server might need to obtain the physical address attributes of each endpoint location to synchronize the network locations between the integrated systems.
7. The third-party mediation server presents the request for new plant assignment to the external plant inventory application.
8. The external plant inventory user assigns the appropriate plant inventory and network locations required for each transmission circuit.
9. The third-party mediation server or the external plant inventory application associates the assigned plant inventory with the MetaSolv Solution circuit ID for future reference and data synchronization processing. It places the plant inventory into a pending assignment status.
10. The third-party mediation server performs a query for the network location ID, if necessary. See "Query for Network Location ID" for more information. The *exportServiceRequestDetailSucceeded* operation provides the absolute endpoints of a circuit; however, the plant engineer can select a path, which requires additional locations to complete the assignment. The internal MetaSolv Solution location IDs are required for the *importPlantAssignment* operation.

11. The third-party mediation server performs a query to obtain physical plant specifications, if necessary. See "Query for Plant Specification ID" for more information. The internal MetaSolv Solution plant specification IDs are required for the *importPlantAssignment* operation.
12. The third-party mediation server performs a query to obtain valid values for plant assignment, if necessary. See "Obtain Valid Values for Plant Import and Export" for more information. The valid values for specific fields are dynamic, depending on table entries maintained by the MetaSolv Solution user. You can obtain these values by executing the operations described in this process.
13. The third-party mediation server sends the *importPlantAssignment* operation for each circuit on the order that requires plant assignments. Plant assignments are required for base circuits; however, they are not required for transmission circuits that ride base circuits. The import operation includes the physical details about the plant transport medium and the network and terminal locations involved in each circuit assignment.

> **Note:** The plant assignment change process might not require a change in cable pair assignments. The third-party plan inventory system must determine whether to assign the same or different cable pair to the circuit based on the status of the cable pair. If plant assignment changes are not required for a circuit, the *importPlantAssignment* operation is not required for the circuit.

14. The MetaSolv Solution Plant API server assigns the plant inventory to the ordered circuits.
15. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
16. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request for Plant Assignment Change

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed a change order for an in-service circuit. The MetaSolv Solution Work Management module places the PA task into Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order, even if it includes multiple circuit items.

2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API Server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.
5. The activity code returned from the *exportServiceRequestDetail* operation is equal to *change* when modifying an existing circuit. The change order within MetaSolv Solution may not require a change to plant facilities. If problems exist with the assigned plant (bad cable pair), the plant inventory assigned to the circuit should be returned to inventory and new plant should be obtained from available inventory.
6. The third-party mediation server performs a query for network location details, if necessary. See "Obtain Network Location Details" for more information. The *exportServiceRequestDetailSucceeded* operation provides the internal MetaSolv Solution location IDs for the absolute endpoints of a circuit. The third-party mediation server might require the physical address attributes of each endpoint location to synchronize the network locations between the integrated systems.
7. The third-party mediation server presents the currently assigned plant for each of the circuits on the change order to the external plant inventory application.
8. The external plant inventory user determines if a change in plant inventory and network locations is required.
9. If a change in plant inventory is not required, the third-party mediation server skips to step 14.
10. The third-party mediation server or the external plant inventory application associates the assigned plant inventory with the MetaSolv Solution circuit ID for future reference and data synchronization processing. It places the plant inventory into a pending assignment status.
11. The third-party mediation server performs a query for the network location ID, if necessary. See "Query for Network Location ID" for more information. The *exportServiceRequestDetailSucceeded* operation provides the absolute end points of a circuit; however, the plant engineer might select a path, which requires additional locations to complete the assignment. The internal MetaSolv Solution location IDs are required for the *importPlantAssignment* operation.
12. The third-party mediation server performs a query for physical plant specifications, if necessary. See "Query for Plant Specification ID" for more information. The internal MetaSolv Solution plant specification IDs are required for the *importPlantAssignment* operation.
13. The third-party mediation server performs a query for valid values, if necessary. See "Obtain Valid Values for Plant Import and Export" for more information. The valid values for specific fields are dynamic, depending on table entries maintained by the MetaSolv Solution user. You can obtain these values by executing the operations described in this process.
14. The third-party mediation server sends the *importPlantAssignment* operation for each circuit on the order that requires plant assignments. Plant assignments are required for base circuits; however, they are not required for transmission circuits that ride base circuits. The import operation includes the physical details about the

plant transport medium and the network and terminal locations involved in each circuit assignment.

> **Note:** For the plant assignment change process, a change in cable pair assignments might be required. The third-party plant inventory system must determine if the same or different cable pair should be assigned to the circuit based on the status of the cable pair. If plant assignment changes are not required for a circuit, the *importPlantAssignment* operation is not required for the circuit.

15. The MetaSolv Solution Plant API server assigns the plant inventory to the ordered circuits.
16. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
17. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request to Cancel Plant Assignment

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed an order for a new circuit, has completed the initial plant assignment task on the provisioning plan, and has canceled the order. The MetaSolv Solution Work Management module places the PA task into Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.
5. The activity code returned from the *exportServiceRequestDetail* operation is equal to Cancel when a new or changed order has been placed, plant assignment activity has started, and the MetaSolv Solution user has canceled the order. This process occurs after the new or change process is complete and before the confirmation process is started.

6. The third-party mediation server identifies all of the outside plant elements related to the circuit or circuits provided in the *exportServiceRequestDetailSucceeded* operation. The third-party mediation server uses the internal MetaSolv Solution circuit ID for each circuit on the order to synchronize the plant elements between the integrated systems.
7. The third-party mediation server presents the currently assigned plant for each of the circuits on the cancel order to the external plant inventory application.
8. The external plant inventory user restores the plant assignments to their previous state prior to the new plant order or change plant order.
9. The third-party mediation server or the external plant inventory application disassociates the assigned plant inventory with the MetaSolv Solution circuit ID and updates the plant inventory status.
10. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
11. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request to Disconnect Plant Assignment

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed an order to disconnect service on a circuit. The MetaSolv Solution Work Management module places the PA task into Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.

   The activity code returned from the *exportServiceRequestDetail* operation is equal to *disconnect* when the MetaSolv Solution user places an order to disconnect a circuit.
5. The third-party mediation server identifies all of the outside plant elements related to the circuit or circuits provided in the *exportServiceRequestDetailSucceeded* operation. The third-party mediation server uses the internal MetaSolv Solution

circuit ID for each circuit on the order to synchronize the plant elements between the integrated systems.

6. The third-party mediation server presents the currently assigned plant for each of the circuits on the cancel order to the external plant inventory application.
7. The external plant inventory user removes the plant assignments and returns them to available inventory.
8. The third-party mediation server or the external plant inventory application places the plant inventory into a pending disconnect status.
9. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
10. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and Plant assignment gateway event.

Request to Cancel Plant Disconnect

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed an order to disconnect service on a circuit, has completed the plant assignment task on the provisioning plan, and has canceled the disconnect order. The MetaSolv Solution Work Management module places the PA task into a Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.

   The activity code returned from the *exportServiceRequestDetail* operation is equal to Cancel when a disconnect order is placed, plant assignment activity has started, and the MetaSolv Solution user has canceled the disconnect order. This process occurs after the disconnect process is complete and before the confirmation process is started.
5. The third-party mediation server identifies all of the outside plant elements related to the circuit or circuits provided in the *exportServiceRequestDetailSucceeded* operation. The third-party mediation server uses the internal MetaSolv Solution

circuit ID for each circuit on the order to synchronize the plant elements between the integrated systems.

6. The third-party mediation server presents the currently assigned plant for each of the circuits on the cancel disconnect order to the external plant inventory application.
7. The external plant inventory user restores the plant assignments to their previous state prior to the disconnect plant order.
8. The third-party mediation server or the external plant inventory application restores the assigned plant inventory with the MetaSolv Solution circuit ID for future reference and data synchronization processing. It places the plant inventory into the previous state prior to the disconnect order.
9. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
10. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request for Change to Due Date

Setup prerequisite: The MetaSolv Solution user created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate Work Management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The Work Management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the Work Management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution user has placed an order, has completed the plant assignment task on the provisioning plan, and has changed the due date on the order. The MetaSolv Solution Work Management module places the PA task into a Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.

   The activity code returned from the *exportServiceRequestDetail* operation is equal to Due Date Change when the MetaSolv Solution user places a new, change, or disconnect order, plant assignment activity has started, and the MetaSolv Solution user has modified the due date for the order. This process occurs after the new, change, or disconnect process is complete and before the confirmation process is started.

5. The third-party mediation server identifies all of the outside plant elements related to the circuit or circuits provided in the *exportServiceRequestDetailSucceeded* operation. The third-party mediation server uses the internal MetaSolv Solution circuit ID for each circuit on the order to synchronize the plant elements between the integrated systems.
6. The third-party mediation server presents the currently assigned plant for each of the circuits on the due date change order to the external plant inventory application.
7. The external plant inventory user updates the planned due date for the pending plant assignments or plant disconnect.
8. The third-party mediation server or the external plant inventory application updates the planned due date.
9. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
10. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request for Plant Assignment Exception

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal a plant assignment request, and the user has associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the circuit identification task or activity and before the circuit design task or activity. The name of the work management task is user defined; however, the example process flow refers to it as the PA task.

Processing prerequisite: The MetaSolv Solution Work Management module places the PA task into Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.

   The activity code returned from the *exportServiceRequestDetail* operation is equal to No Activity when the MetaSolv Solution user has placed a new, change, or disconnect order; however, analysis of the requested circuit or circuits indicates no plant inventory is required to fulfill the order.

This order activity code should not occur in normal situations; however, the third-party mediation server should prepare for it. If this order activity is received for a circuit, no action is required for the specific circuit item.

5. The third-party mediation server determines that no plant assignment activity is required.
6. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server after plant assignments are complete for all of the ordered circuit items.
7. The Work Management module changes the PA task to complete. This automatic activity relies on the user to correctly define the PA task and plant assignment gateway event.

Request to Complete Plant Assignment

Setup prerequisite: The MetaSolv Solution user has created a gateway event to signal the due date task is complete. The user has also associated the gateway event with the appropriate work management task. The gateway event name must be coordinated between the system integrator and the MetaSolv Solution user. The work management task must be placed after the due date task. The name of the work management task is user defined; however, the example process flow refers to it as the plant assignment complete task.

Processing prerequisite: The MetaSolv Solution user has placed an order and has completed all work management tasks through the due date task. The MetaSolv Solution Work Management module places the PAC task into Ready status, determines if the gateway event rules are satisfied, and sends the signal to the gateway event server.

This gateway event is sent to the mediation server when the plant assignments are complete, all other equipment and facility assignments are complete, and the circuit items are either in service or disconnect as requested by the order.

This gateway event should not be sent when the MetaSolv Solution user cancels the original order. However, gateway event signals are largely user-controlled, so the third-party mediation server should be prepared for such an event.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API Server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.
5. The third-party mediation server identifies all of the outside plant elements related to the circuit or circuits provided in the *exportServiceRequestDetailSucceeded* operation. The third-party mediation server uses the internal MetaSolv Solution circuit ID for each circuit on the order to synchronize the plant elements between the integrated systems.

6. The third-party mediation server sends the *exportPlantAssignment* operation to the MetaSolv Solution Plant API server for each circuit to confirm the plant assignments on the external plant inventory application are consistent with the plant assignment in the MetaSolv Solution application. This is an important step because MetaSolv Solution users have the option to reject the plant assignments made with the *importPlantAssignment* operation.
7. The MetaSolv Solution Plant API server returns the *exportPlantAssignmentSucceeded* operation to the third-party mediation server with the list of transmission circuits and the actual plant assignments.
8. The third-party mediation server presents the pending plant assignments or pending plant disconnects for each of the circuits on the order to the external plant inventory application.
9. The external plant inventory user completes the assignment or disconnect actions.
10. The third-party mediation server or the external plant inventory application updates the plant inventory status.
11. The third-party mediation server sends an *eventCompleted2* operation to the MetaSolv Solution gateway event server.
12. The Work Management module changes the PAC task to complete. This automatic activity relies on the user to correctly define the PAC task and plant assignment complete gateway event.

Import Plant Assignment Failed

Depending on the activities required by the external plant inventory application or the interaction between the third-party mediation server and the *external plant inventory application*, it might not be possible for the third-party mediation server to successfully import plant assignment information for all of the circuits requested. If this situation occurs, the third-party mediation server can implement the following process.

1. The MetaSolv Solution gateway event server sends the *eventOccurred* operation to notify the third-party mediation server that plant assignment activity is required for a specific service order. One gateway event signal is sent for the entire service order even if it includes multiple circuit items.
2. The third-party mediation server sends an *eventInProgress2* operation to the MetaSolv Solution gateway event server.
3. The third-party mediation server sends a *exportServiceRequestDetail* operation to the MetaSolv Solution Plant API server for the order.
4. The MetaSolv Solution Plant API Server returns the *exportServiceRequestDetailSucceeded* operation to the third-party mediation server with the list of transmission circuits, which require plant assignment action, and the activity code, which indicates the plant assignment action to take.
5. The third-party mediation server sends the *importPlantAssignment* operation for each circuit on the order that requires plant assignments.
6. The MetaSolv Solution Plant API server returns the *importPlantAssignmentFailed* operation with an error operation and a list of circuits, for which the import was successful.
7. The third-party mediation server, the external plant inventory application, and/or the external plant inventory application user are unable to resolve the import error.

8. The third-party mediation server sends the *eventFailed2* operation to the MetaSolv Solution gateway event server.

Obtain Network Location Details

The process provides the ability to obtain details about a specific network location based on the internal MetaSolv Solution location ID.

1. The third-party mediation server sends a *getLocation* operation to the MetaSolv Solution Infrastructure API server to obtain the physical address related to the specific internal MetaSolv Solution location ID.
2. The MetaSolv Solution Infrastructure API server returns *getLocationSucceeded* operation with the physical addresses and other network location data.

Query for Network Location ID

This process provides the ability to query for and derive the internal MetaSolv Solution location ID for a specific network location based on the physical address or other defining attributes.

1. The third-party mediation server prepares to send the *queryNetworkLocations_v2* operation to the MetaSolv Solution Infrastructure server to obtain the internal MetaSolv Solution network location ID based on one or more physical address components.

   A data structure is provided with this operation to specify search criteria. Knowledge of The MetaSolv Solution structure formats is critical to specifying physical address search criteria. Name and value pairs for physical address components are user-defined. The names of the physical address components must be coordinated between the MetaSolv Solution user and the systems integrator.
2. The third-party mediation server, optionally, sends the *getNetworkLocationTypes* operation to the MetaSolv Solution Infrastructure server to obtain the list of network location types, which can be used as search criteria for the *queryNetworkLocations_v2* operation.
3. The MetaSolv Solution Infrastructure server returns the *getNetworkLocationTypesSucceeded* operation with the list of internal MetaSolv Solution network location type IDs along with their descriptions.
4. The third-party mediation server, optionally, sends the *getNetworkLocationCategories* operation to the MetaSolv Solution Infrastructure server to obtain the list of network location categories, which can be used as search criteria for the *queryNetworkLocations_v2* operation.
5. The MetaSolv Solution Infrastructure server returns the *getNetworkLocationTypesSucceeded* operation with the list of internal MetaSolv Solution network location category IDs along with their descriptions.
6. The third-party mediation server sends the *queryNetworkLocations_v2* operation to the MetaSolv Solution Infrastructure server to obtain the internal MetaSolv Solution network location ID based on one or more physical address components provided in the search criteria.
7. The MetaSolv Solution Infrastructure server sends the *queryNetworkLocationsSucceeded* operation to the third-party mediation server with the internal MetaSolv Solution network location ID.

Query for Plant Specification ID

This process provides the ability to query for and derive the internal MetaSolv Solution plant specification ID for a specific type of outside plant transport facilities.

1. The third-party mediation server sends *queryPlantTransportPhysicalCompositionSpec* operation to obtain the internal MetaSolv Solution plant transport physical composition specification code. In the query operation, one or more query values can be specific to limit the size of the result set.
2. The MetaSolv Solution Plant API server returns the *queryPlantTransportPhysicalCompositionSpecSucceeded* operation with all of the plant transport specifications, which meet the query criteria. The plant element specifications identify the physical properties of a plant transport medium such as cable, fiber, or air.

Obtain Valid Values for Plant Import and Export

For specific fields used in the plant import process, the valid values are dynamic. The operations included in this process provide the ability to export the valid values. There are many different ways to implement these operations. An integrator can choose to export the valid values each time he prepares the data for the import operation. An integrator may also choose to export the valid values at fixed intervals or during the server start-up routine and store the values for later reference.

1. The third-party mediation server sends the *getFunctionCodes* operation to the MetaSolv Solution Plant API server to obtain a list of valid function codes. Examples of function codes include transmit and receive.
2. The MetaSolv Solution Plant API server returns the *getFunctionCodesSucceeded* operation with the valid function codes.
3. The *getLoadingTypes* operation can be used to obtain a list of valid loading types. One of these loading types (NL for non-loading) can be passed in the *queryPlantTransportPhysicalCompositionSpec* operation to limit the list of plant transport physical composition specifications to a specific loading type.
4. The MetaSolv Solution Plant API Server returns the *getLoadingTypesSucceeded* operation with the valid loading types.
5. You can use the *getPlantTransportClasses* operation to obtain a list of valid classes. One of these classes (copper, fiber, microwave, or satellite) can be passed in the *queryPlantTransportPhysicalCompositionSpec* operation to limit the list of plant transport physical composition specifications to a specific class.
6. The MetaSolv Solution Plant API Server returns the *getPlantTransportClassesSucceeded* operation with the valid transport classes.
7. The *getPlantElementAssignmentStatuses* operation can be used to obtain a list of cable pair or fiber status codes.
8. The MetaSolv Solution Plant API server returns the *getPlantElementAssignmentStatusesSucceeded* operation with the status codes and definitions.

9

The PSR Ancillary API

The PSR Ancillary API provides a bi-directional interface for LIDB/CNAM information and E911 data in National Emergency Numbering Association (NENA) format for transfer to the area E911 provider. All fields needed for NENA 2.1 compliancy are provided by this API.

Implementation Concepts

The PSR Ancillary API only supports batch-mode interaction. Two types of operations are provided for this purpose: extract and respond. The extract operation performs a data export and the respond operation performs data import. The only minor variation is the appearance of operations of the form extract{Function}Empty in the WDINotification interface. Such notification callback operations are invoked by PSRAncillaryServer to indicate the absence of candidate extract records for that day.

Essential Terminology

Table 9–1 lists the terms that identify information and concepts that are required to understand the PSR Ancillary API.

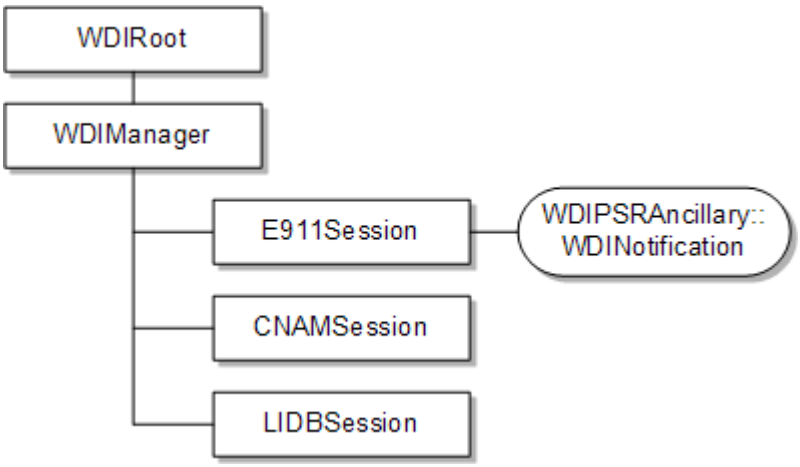
Table 9–1 PSR Ancillary API Essential Terminology

Term	Definition
CNAM	A telephone service used to display the name and telephone number of the caller.
E911	A telephone service used to provide emergency (911) operators with the caller's telephone number and location.
LIDB	A telephone service used to verify a telephone number for toll service and third-party billing, such as for validation of calling card numbers.

PSR Ancillary API Interfaces

Figure 9–1 illustrates the relationship of the interfaces in the PSR Ancillary API.

Figure 9–1 PSR Ancillary API Interfaces



E911Session Interface Operations

Table 9–2 lists the operations available in the E911Session of the **PSRANCILLARY.IDL** file.

Table 9–2 E911Session Interfacer Operations

Operation	WDINotification
extract	extractE911Succeeded extractE911Failed extractE911Empty
respond	respondE911Succeeded respondE911Failed

CNAMSession Interface Operations

Table 9–3 lists the operations available in the CNAMSession of the **PSRANCILLARY.IDL** file.

Table 9–3 CNAMSession Interface Operations

Operation	WDINotification
extract	extractCNAMSucceeded extractCNAMFailed extractCNAMEmpty
respond	respondCNAMSucceeded respondCNAMFailed

LIDBSession Interface Operations

Table 9–4 lists the operations available in the LIDBSession of the **PSRANCILLARY.IDL** file.

Table 9–4 LIDBSession Interface Operations

Operation	WDINotification
extract	extractLIDBSucceeded extractLIDBFailed extractLIDBEmpty
respond	respondLIDBSucceeded respondLIDBFailed

Implementation Concepts

All interfaces and operations for the PSR Ancillary API are defined in WDIPSRANCILLARY.IDL. Some interfaces and operations are implemented in the PSR Ancillary API server, PSRAncillaryServer, and the others must be implemented in your application.

The PSR Ancillary API supports only batch-mode interaction. Two types of operations are provided for this purpose: extract and respond. The extract performs a data export and the respond operation performs data import. The only minor variation is the appearance of operations of the form extract{Function}Empty in the WDINotification interface. Such notification callback operations are invoked by PSRAncillaryServer to indicate that there were no candidate extract records for that day.

The PSR Ancillary API and Smart Tasks

The PSRAncillary server E911 functionality does not use the Gateway Event Model. An E911 Smart Task is used to create the row data and determine function codes. The PSRAncillary server is now responsible for extracting the row data with the PSRAncillary extract method and matching the extract data row to the response records returned in the respond method call. The WDISignal interface is no longer used. E911 Gateway Events still show up in the GUI and need to be removed from the provisioning plans by whoever is responsible for modifying provisioning plans.

Note: Do not use Gateway Events for E911.

Field by Field Matching Between Extract Row and Response Record

The telephone number data in the database, which was sent out in the E911ExtRecord, is matched to the data sent back for every calling telephone number (CTN) (911RespRecord.ctn) in the respond method call. Every field sent back in the E911RespRecord must exactly match the row data in the MetaSolv Solution database for any record marked Complete (E911RespRecord.sti = ‘C’.) If a match does not occur the E911 response code is changed to error and any mismatched fields are displayed in the GUI.

Since this is a database matching system, the PSRAncillary server considers any data field changes an error, but a succeeded notification is sent back to the gateway vendor so the errors can be seen, corrected, and resent using the MetaSolv Solution GUI. Since the CTN is the key field triggering matching, if any CTN cannot be matched to a record in Sending status for an extractSeq, a failed notification is sent back to the gateway vendor with error messages for all CTN fields not matched to a Sending telephone number in the database.

The gateway vendor is responsible for removing the offending calling telephone numbers and calling another respond method. It is impossible for the PSRAncillary server to determine if the database information is correct without matching a telephone number in Sending status to a valid CTN. It is the gateway vendor's responsibility to contact the appropriate party (either the customer or the third-party provider) to correct any discrepancies between a CTN and the telephone numbers in the MetaSolv Solution database.

Rules of Operation

1. Each extract and respond operation has an extractID parameter. This extractID is generated by the third-party application and is transferred back to your application as a parameter on the succeeded, failed and empty notifications for the extract and respond. The extractID provides the third-party application with a mechanism to match a notification with the corresponding extract or respond request.
2. When an extract operation is invoked by your application, MetaSolv Solution generates and returns a unique sequence number in the notify callback. This sequence number is passed in the extractSeq parameter. When your application sends the response for the extract, the extractSeq in the response structure must be the same as the extractSeq sent in the corresponding extract structure. The extractSeq provides the APIs with a mechanism to match an extract with a response. The third-party application is responsible for managing database transaction processing. This means the third-party application establishes the MetaSolv Solution database connection through the WDITransaction interface of the API and controls commit and rollback processing.

Each extract and respond structure has an extractSeq parameter. The extractSeq is generated by MetaSolv Solution and sent to the third-party application through the extract operation. When the third-party application sends the respond for the extractSeq in the E911 Update respond structure, it must be the same as the extractSeq sent in the corresponding extract.

The extractSeq provides a mechanism to match an extract with a response.

1. There is an *extractID* parameter for each extract and respond operation. This *extractID* is generated by the third-party application and is transferred back to your application as a parameter on the successful, failed, and empty notifications for the extract and respond. The *extractID* provides the third-party application with a mechanism to match a notification with the corresponding extract or respond request.
2. When an *extract* operation is invoked by your application, MetaSolv Solution generates and returns a unique sequence number in the notify callback. This sequence number is passed in the extractSeq parameter. When your application sends the response for the extract, the extractSeq in the response structure must be the same as the *extractSeq* sent in the corresponding extract structure. The extractSeq provides the APIs with a mechanism to match an extract with a response.

The PSR Ancillary API defines the interfaces between MetaSolv Solution and a third-party gateway. This API facilitates the sharing of E911, CNAM, and LIDB information between MetaSolv Solution and the appropriate database provider.

The third-party application is responsible for managing database transaction processing. This means the third-party application establishes the MetaSolv Solution database connection through the WDITransaction interface of the API and controls commit and rollback processing.

There is an extractID parameter for each extract and respond operation. This extractID is generated by the third-party application, then transferred back to the third-party application as a parameter on the succeeded, failed, and empty notifications for the extract and respond. The extractID provides the third-party vendor with a mechanism to match a notification with the corresponding extract or respond request.

There is an extractSeq parameter within each extract and respond structure (extractE911, respondE911, extractCNAM, respondCNAM, extractLIDB, and respondLIDB). The extractSeq is generated by MetaSolv Solution and sent to the third-party application through the extract operation. When the third-party application sends the respond for the extractSeq, in the respond structure, it must be the same as the extractSeq sent in the corresponding extract. The extractSeq provides a mechanism to match an extract with a response.

Extract Sequence Matching

The E911Extract.extractSeq sent in the succeeded notification must be returned in the E911Response.extractSeq in the respond method call. The extractSeq field allows database rows in a Sending status to be matched to records in the E911RespRecords array (see Field by Field Matching Between Extract Row and Response Record.) The E911 Update extactSeq field allows multiple respond methods to be processed simultaneously (if the extractSeq are different.) The respond method can be made any time after the extract method is called, and can contain a partial set of records. The response does not have to contain every record extracted previously if those records contain the correct extractSeq. This allows the PSRAncillary server to match records using unordered extract and respond method calls.

Extract method calls and/or respond method calls can be done daily, in any amount, in any order. There are restrictions enforced by the PSRAncillary server as to how the method calls are processed. If an extract method call has not finished processing before another extract method call is received, the second (and subsequent) extract method calls wait until the first has finished processing before being allowed to do further processing. If a respond method call for an extractSeq is not finished processing before another respond method call using the same extractSeq is received, the second (and subsequent) respond method calls with the same extractSeq wait until the first has finished processing before being allowed to do any further processing.

Note: The previous restriction of a twenty-four hour turn around time using one extract method call followed by one respond method call containing every record extracted is no longer required.

Extract and Respond Scenario

Rules for extract and response scenarios are summarized in the following list:

- Extractseq matching on extract and respond is strictly enforced.
- Extract and respond does not need a 24-hour turn around.
- Respond does not have to follow extract (which means multiple extracts can be done before a respond is sent back.
- Respond does not have to contain all the records extracted for the extractSeq on the extract.
- Respond cannot contain telephone numbers (matched by CTN) that were not sent in the extract or were previously processed by another respond with the same extractSeq (no unmatched, if a telephone number is sent in the response that

cannot be matched to a telephone number in sending status, the respond method fails until the CTN is removed).

- Extract cannot have telephone numbers differing only by suffix (the CTN would be the same which makes matching the CTN on the respond impossible), the CTNs are errored but the extract still occurs with the offending telephone numbers removed.
- All fields are strictly matched in the respond, if any field sent in the extract has changed, the record is errored but the respond is still considered a success.

> **Note:** Previously, customers set the **exd** field, which is stamped on the extract record. This field should not be modified. The only supported function codes are C or E Matching for STI (status indicator or response code).

Error Logging Changes

PSRAncillary server E911 errors are now logged to the E911.log file. If the error is related to a data issue and can be corrected in the GUI, the error also appears in the PSRAncillary Maintenance window. Errors displaying in the GUI include NENA errors returned in the response, data integrity errors that can be corrected in the GUI, or in the determination of the ordering of extract records. Errors written only to the E911.log file are fatal server errors needing immediate attention. These errors cannot be fixed using the GUI alone.

Any server error appearing in the PSRAncillary Maintenance window also appears in the E911.log file, but not all errors written to the E911.log file appear in the PSRAncillary Maintenance window. Both the E911.log file and the PSRAncillary Maintenance window must be used to monitor errors logged by the server.

The PSRAncillary E911 error logging includes the following:

- Logging the PSRAncillary server errors to the E911.log file in the path set up in the gateway.ini file. Any previous log files used by the PSRAncillary server (`MetaSolv Solution`.log/appserver.log, WDI.log, EventServer.log, and PSR Ancillary.log) no longer contain any E911 related error logging (although CNAM and LIDB information is still logged to these log files.)
- Errors written to both the E911.log file and the MetaSolv Solution database E911 error table (writing to the error table allows the errors to be seen in the PSRAncillary Maintenance window.) Scenarios where this is most likely to occur include but are not limited to:
  - Duplication of telephone numbers differing only by suffix on the extract database table with an extract indicator set to **Y**.
  - Respond record returning with an **E** in the **sti** field, indicating a NENA error. The NENA error returned is written to the E911 error table with a prepended 'NENA Error:' stamp on the error message.
  - Respond record returning with a **C** in the **sti** field and the record does not pass field matching criteria. An error is written for each field that does not match up with a prepended **MSLV Error:** stamp on the error message.
  - Updating an extract database row while the E911 record is in 'Sending' status.
- All E911 error rows are cleared when the gateway provider calls the extract method on the PSRAncillary server. If the PSRAncillary server is not used as part of the E911 solution, the E911 rows are never deleted from the database.

If the System Queue is used to complete the task, errors can be written to the WM_TskSv.log file and to the SERVER_LOG database table. An error written to the SERVER_LOG table is displayed in the GUI. The PSRAncillary server logging occurs independent of System Queue logging. Therefore, errors occurring in the PSRAncillary server are logged by the PSRAncillary server and they can also be logged by the System Queue. As a general rule, if an error occurs on the PSRAncillary server, it is logged normally by the PSRAncillary server and then picked up by the System Queue and logged on the SERVER_LOG table so it can be displayed in the GUI. If the error is not written to the E911.log, then the error most likely occurred due to functional issues on the System Queue, and is logged in the WM_TskSv.log file and possibly the SERVER_LOG table (if the error is of a nature that needs to be displayed by the GUI.)

Process Flow

This section contains a sample process flow for an unsolicited message. Use the sample flow as a template for developing your own process flows.

Unsolicited Messages

When the message is initiated by the third party (unsolicited), MetaSolv Solution plays the role of the server, and the third-party application plays the role of the client. Unsolicited messages are processed asynchronously, meaning a callback mechanism is used to report back the results of an operation invoked by the third-party application.

Table 9–5 lists the interfaces and operations that MetaSolv Solution implements using IDL files provided with the PSR Ancillary API.

Table 9–5 PSR Ancillary API Operations

Interface	Operations
WDIRoot	connect disconnect
WDIManager	startE911Session startCNAMSession startLIDBSession destroyE911Session destroyCNAMSession destroyLIDBSesion startTransaction destroyTransaction startSignal destroySignal
E911Session	extract respond
CNAMSession	extract respond
LIDBSession	extract respond

Table 9–6 lists the interfaces and operations that your application can implement.

Table 9–6 PSR Ancillary API Notification Operations

Interface	Operations
WDINotification	extractE911Succeeded
	extractE911Failed
	extractE911Empty
	respondE911Succeeded
	respondE911Failed
	extractCNAMSucceeded
	extractCNAMFailed
	extractCNAMEmpty
	respondCNAMSucceeded
	respondCNAMFailed
	extractLIDBSucceeded
	extractLIDBFailed
	extractLIDBEmpty
	respondLIDBSucceeded
	respondLIDBFailed

Sample Unsolicited Message Process Flow

1. The third-party application binds to the PSR Ancillary server to get a WDIRoot object reference.
2. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.
3. The third-party application invokes the *startE911Session*, *startCNAMSession* or *startLIDBSession* operation of the WDIManager interface to get an E911Session, CNAMSession, or LIDBSession object reference, respectively.
4. The third-party application instantiates a WDINotification object.
5. The third-party application invokes the *startTransaction* operation of the WDIManager interface, which yields a WDITransaction object reference. This object reference passes as a parameter on subsequent operations and is used by the third-party application upon completion of processing to initiate the *commit* or *rollback* operation.
6. The third-party application invokes the appropriate operation on the session object reference returned in step 3 (that is, *extract* or *respond*). The WDINotification and WDITransaction object references are passed as parameters.
7. The PSR Ancillary server processes the invoked operation of the session object and invokes the appropriate failed, succeeded, or empty operation of the input WDINotification upon completion. The empty notification is used only for the extract operations.
8. If the failed operation was invoked in step 7, the third-party application initiates the *rollback* operation of the WDITransaction interface.
9. If the *succeeded* or *empty* operations were invoked in step 7, the third-party application initiates the *commit* operation of the WDITransaction interface.

10. The third-party application invokes the appropriate destroy session operation of the WDIManager interface (*destroyE911Session*, *destroyCNAMSession* or *destroyLIDBSession*).
11. The third-party application invokes the *destroyTransaction* operation of the WDITransaction interface.
12. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

Auto Respond Preference

The auto respond functionality is now available to those who want to receive responses from a third-party provider in a format other than the electronic respond method provided by the PSRAncillary server. If a customer receives an error from a third-party provider, in the form of a fax, email, letter, or other method, the PSRAncillary Maintenance window is used to adjust the record and resend it to the gateway vendor. The respond method cannot be used by the gateway vendor to respond to records after the extract method is called, this causes a failed notification to be sent to the gateway vendor. The third-party provider must be informed to send ALL NENA errors directly to the customer.

When this preference is turned on, all records sent to the gateway vendor are immediately marked as Complete as if a successful electronic response was received. The auto respond preference is not a mix and match preference. The preference is either turned on and no electronic responses are received using the respond method call, or the preference is turned off and all records need to receive an electronic response using the E911 Update PSRAncillary server respond method. The server must be restarted after the new lines below are added to the gateway.ini file. This registers the new preference with the PSRAncillary server. Add the following lines to the gateway.ini exactly as they appear below to turn the preference on:

```
[PSRAncillary]
AutoRespond=true
```

Glossary of Terms and Abbreviations

- ALI: Automatic Location Identification; automatic display at the PSAP of the caller's telephone number, the address/location of the telephone, and supplementary emergency services information.
- ALI database: The set of ALI records residing on a computer system.
- E911: Used to refer to emergency 9-1-1.
- E911 information: Any data that is captured by MetaSolv Solution and sent to the E911 Service Provider by the gateway vendor. This data is any information in MetaSolv Solution that needs to be formatted into a NENA specific transfer protocol in order to be put into an ALI database.
- E911 record: Collectively refers to any data passed from MetaSolv Solution through the gateway vendor to the E911 service provider. The data can be in the form of a database row, a JAVA object or a NENA specified transfer format The data representation may be different but the E911 information remains consistent unless a valid modification by a system occurs.
- E911 Service Provider: System responsible for storing and maintaining E911 records on an ALI database and makes the information available to the PSAPs. The gateway vendor forwards the E911 records to the E911 service provider, any

reference to the E911 service provider should be considered a System as defined by the UML specification.

- E911 Smart Task: The task on a provisioning plan giving the user control of the information that is sent to the E911 service provider.
- Gateway Vendor: The system using the PSRAncillary server to obtain E911 information. The gateway vendor is responsible for taking the E911 information and formatting it into the appropriate NENA transfer protocol and passing the E911 information to the appropriate E911 service provider. Any reference should be considered an actor to MetaSolv Solution as defined by UML specifications. In this case, the actor is also a system.
- MetaSolv Solution E911 Administrator: The person responsible for working E911 related issues in MetaSolv Solution. This person has domain knowledge about NENA, emergency 9-1-1 processing, use of MetaSolv Solution to work E911 related issues and tasks. Any reference should be considered an actor to MetaSolv Solution as defined by UML specifications. The MetaSolv Solution E911 administrator can be considered a more specialized role than that of the MetaSolv Solution user.
- MetaSolv Solution Database: Used to persist information for MetaSolv Solution.
- MetaSolv Solution User: The person using MetaSolv Solution to do order entry or location/customer maintenance. This person is not expected to know specific E911 related information other than the very basics needed in an order entry capacity.
- NENA: National Emergency Number Association; organization responsible for standardizing emergency 9-1-1 procedures.
- PSAP: Public Safety Answering Point; a facility equipped and staffed to receive 9-1-1 calls.
- PSRAncillary server: The MetaSolv Solution API server handling the E911 processing methods of the WDIPSRAncillary IDL interface. This term is used when referring to the server portion of MetaSolv Solution.

10

The PSR Order Entry API

The Product Service Request (PSR) module integrates telephone number administration, product catalog, and customer management with an ordering engine. It captures and stores information required to reference and provision a service request. Other modules in Oracle Communications MetaSolv Solution rely upon the information in the PSR module to enable fulfillment of an order for a product or service for a specific customer. The service request itself initiates several other processes, such as service design and provisioning, telephone number assignment, directory services, LIDB/CNAM, and E911. The PSR module is an ordering engine that enables you to order and provision telephone and non-telephone products including dial tone services, centrex, ISDN-BRI, ISDN-PRI, private line circuits, ATM/frame services, travel cards, customer premise equipment, etc. Any product you define in the product catalog can be ordered through the PSR module.

The PSR Order Entry API provides access to necessary data and business rules underlying the order management functionality in the PSR module, which allows orders to be provisioned. The PSR Order Entry API enables users to enter order information in other systems, bypassing the data entry and data management functionality of the PSR module. The other system is responsible for order entry and validation of information that allows products to be provisioned. The other system is also responsible for the initiation of the mediation layer and/or API. The API inserts into the MetaSolv Solution database the customer account, service location and product service request information necessary for any telecom products or services that are provisioned. The functionality of the other MetaSolv Solution modules remains intact and references the service request information in the same way as is done for a service request that is entered through the PSR GUI.

The PSR Order Entry API provides IDL for PSR ordering, enabling, retrieval, creation, updating, and deletion of PSR orders. The structure of the PSR API architecture is based on the following assumptions:

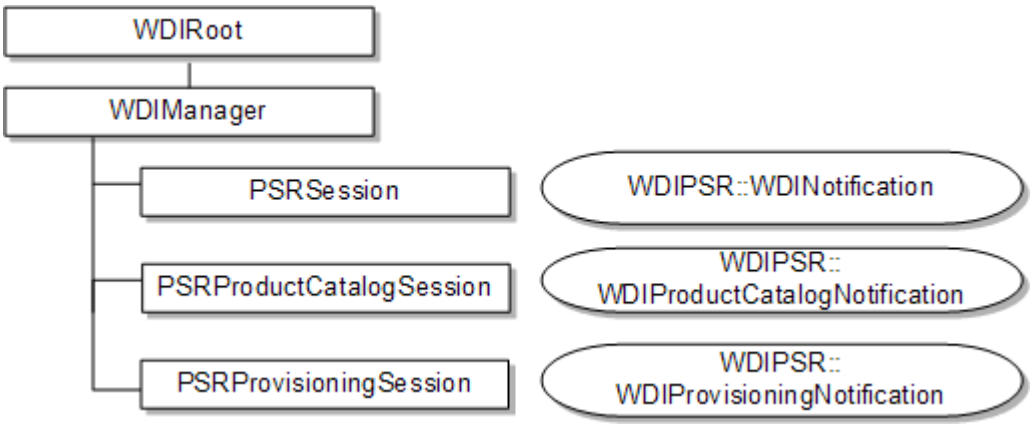
- Your application is responsible for following MetaSolv Solution business rules. These business rules include:
  - All static values are defined in the IDL files as ENUM types. MetaSolv Solution does not provide valid values for user-defined values.
  - Any customer can be exported from the database. The status is not checked when exporting customers.
  - Customers can be exported based on the customer ID.
  - If an error is encountered at any point, the individual operation fails but the process can continue.
  - Import of an updated customer can be performed only on a customer that currently exists in the database.

- The third-party application is responsible for managing all database transactions, including *commit* and *rollback* processing.

PSR Order Entry API Interfaces

Figure 10–1 shows the relationship of the interfaces in the PSR Order Entry API.

Figure 10–1 PSR API Session Interfaces



WDIManager Interface

Table 10–1 lists the operations in the WDIManager interface of the WDIPSR.IDL file and their accompanying description or notification operations.

Table 10–1 PSR Order Entry API WDIManager Interface Operations

Operation	Description
startPSRSession	Obtains the object reference of the PSRSession
destroyPSRSession	Terminates the PSRSession
startPSRSession2	Obtains the object reference of the PSRSession2
destroyPSRSession2	Terminates the PSRSession2
startPSRProductCatalogSession	Obtains the object reference of the PSRProductCatalogSession
destroyPSRProductCatalogSession	Terminates the PSRProductCatalogSession
startPSRProvisioningSession	Obtains the object reference of the PSRProvisioningSession
destroyPSRProvisioningSession	Terminates the PSRProvisioningSession
startInfrastructureSession	Obtains the object reference of the InfrastructureSession
destroyInfrastructureSession	Terminates the InfrastructureSession
startTransaction	commit rollback
destroyTransaction	Terminates the Transaction

Table 10–1 (Cont.) PSR Order Entry API WDIManager Interface Operations

Operation	Description
startSignal	eventOccurred eventTerminated eventInProgress eventCompleted eventErrored
destroySignal	Terminates the Signal
startInSignal	eventInProgress eventCompleted eventErrored
destroyInSignal	Terminates the Insignal

PSRSession Interface Operations

Table 10–2 lists the operations in the PSRSession of the WDIPSR.IDL file and their accompanying notification operations.

Table 10–2 PSRSession Interface Operations

Operation	WDINotification
assignCFA_v2	assignCFASucceeded_v2 assignCFAFailed_v2
exportAccountProvisioningData_v2	accountProvisioningDataExportSucceeded_v2 accountProvisioningDataExportFailed
exportAccountServerDataPSR_v2	accountServerDataPSRExportSucceeded_v2 accountServerDataPSRExportFailed
exportAllServiceLocations_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	PSRExportFailed
exportCFAInfo_v2	exportCFAInfoSucceeded_v2 exportCFAInfoFailed_v2
exportCLLILocation_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	exportCLLILocationSucceeded_v2 exportCLLILocationFailed
exportCPNIConsents_v2	PSRexportCPNIConsentsSucceeded_v2 PSRexportCPNIConsentsFailed
exportCreditCardAuthorizationData_v2	creditCardAuthorizationDataExportSucceeded_v2 creditCardAuthorizationDataExportFailed
exportCustomerAccount_v2	PSRCustomerExportSucceeded_v2 PSRExportFailed
exportCustomerAccount_v3	PSRCustomerExportSucceeded_v3 PSRExportFailed
exportCustomerAccounts_v2	PSRCustomerExportSucceeded PSRExportFailed

Table 10–2 (Cont.) PSRSession Interface Operations

Operation	WDINotification
exportCustomerAccounts_v3	PSRCustomerExportSucceeded PSRExportFailed
exportCustomerOrders_v2 - Deprecated.	PSRExportCustomerOrdersSucceeded_v2
exportCustomerOrders_v3	PSRExportCustomerOrdersSucceeded_v3
exportCustServiceLocations_v2	PSRExportFailed
exportDomainNameRegistrationData_v2	domainNameRegistrationDataExportSucceeded_v2 domainNameRegistrationDataExportFailed
exportEMailProvisioningData_v2	emailProvisioningDataExportSucceeded_v2 emailProvisioningDataExportFailed
exportEMailServerDataPSR_v2	emailServerDataPSRExportSucceeded_v2 emailServerDataPSRExportFailed_v2
exportFTPServerDataPSR_v2	FTPServerDataPSRExportSucceeded FTPServerDataPSRExportFailed_v2
exportNGNPSR	exportNGNPSRSucceeded exportNGNPSRNoDataFound exportNGNPSRFailed
exportNGNServItem	exportNGNServItemSucceeded exportNGNServItemFailed
exportOrder_v2 - Deprecated.	PSRExportOrderSucceeded_v2
exportOrder_v3	PSRExportOrderSucceeded_v3
exportOrderValues_v2	PSROrderValuesExportSucceeded_v2
exportProvisioningData_v2	provisioningDataExportSucceeded_v2 provisioningDataExportFailed
exportPSR_v2 - Deprecated.	exportPSRSucceeded_v2 exportPSRFailed
exportPSR_v3	exportPSRSucceeded_v3 exportPSRNoDataFound_v3 exportPSRFailed_v3
exportServiceLocations_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	PSRSvcLocationExportSucceeded_v2 PSRExportFailed
exportServItem_v2 - Deprecated.	PSRServItemSucceeded_v2 PSRExportFailed
exportServItem_v3	PSRServItemSucceeded_v3 PSRExportFailed
exportWebHostingProvisioningData_v2	webHostingProvisioningDataExportSucceeded_v2 webHostingProvisioningDataExportFailed
exportWebHostingServerDataPSR_v2	webHostingServerDataPSRExportSucceeded webHostingServerDataPSRExportFailed_v2

Table 10–2 (Cont.) PSRSession Interface Operations

Operation	WDINotification
getOrderStatus_v2	getOrderStatusSucceeded_v2 getOrderStatusFailed_v2
importCPNIConsent_v2	PSRImportCPNIconsentSucceeded PSRImportCPNIconsentFailed
importNewCustomerAccount_v2	PSRImportSucceeded PSRImportFailed
ImportNewCustomerAccount_v3	PSRImportSucceeded PSRImportFailed
importNewServiceLocation_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	PSRImportSucceeded PSRImportFailed
importNGNPSR	importPSROrderSucceeded importPSROrderFailed
importPSR_v3	importPSROrderSucceeded_v3 importPSROrderFailed_v3
importPSROrder_v2 - Deprecated.	importPSROrderSucceeded importPSROrderFailed
importUpdatedCustomerAccount_v2	PSRImportSucceeded PSRImportFailed
ImportUpdatedCustomerAccount_v3	PSRImportSucceeded PSRImportFailed
importUpdatedServiceLocation_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	PSRImportSucceeded PSRImportFailed
processBillingTelephoneNumber_v2	processBillingTelephoneNumberSucceeded_v2 processBillingTelephoneNumberFailed
searchMsag_v2	searchMsagSucceeded_v2 searchMsagFailed
surveyUpdate_v2	surveyUpdateImpactSucceeded_v2 surveyUpdateImpactFailed
unassignCFA_v2	unassignCFASucceeded_v2 unassignCFAFailed_v2
updateDomainProvisioning_v2	updateDomainProvisioningSucceeded_v2 updateDomainProvisioningFailed
validateCustomerAccount_v2	validateCustomerAccountSucceeded_v2 validateCustomerAccountFailed
validateCustomerAccount_v3	validateCustomerAccountSucceeded_v3 validateCustomerAccountFailed
verifySvcLoc_v2 - Deprecated. Use the corresponding Infrastructure API operation instead.	verifySvcLocSucceeded verifySvcLocFailed

PSRSession Operation Descriptions

This section describes the operations defined in the WDIPSR.IDL file.

- importNGNPSR

  Enables import of orders with template-based service items (those with item types of System, Element, Connector, or Equipment) and nontemplate-based service items.

- exportNGNPSR

  Enables export of orders with template-based and nontemplate-based service items. This export includes header information, such as customer identification information.

- exportNGNServItem

  Enables export of template-based and nontemplate-based service items. This export includes service items only; no header information is exported. Using this operation, you can export all service items for a customer.

PSR Order Entry API Preferences

MetaSolv Solution defines several preferences that are specific to the PSR API. These preferences are located in the Preferences window, in the **API/PSR Order Entry API Preferences** directory.

Bypass PSR API Switch Validation for TN assignment

This system level preference is a check box that can be deselected (N) or selected (Y):

- N: The PSR API switch validation *is* performed for a TN Assignment.
- Y: The PSR API switch validation is *not* performed for a TN Assignment.

Bypass Selected PSR API Import Structure Validation

This system level preference is a check box that can be deselected (N) or selected (Y):

- **N:** The software does not bypass any PSR API import structure validation.
- **Y:** Selected PSR API validation is not as strict. This functionality currently exists only for values and label validations. When a structure fails validation, it is not processed by the APIs and the incorrect information is not entered into the MetaSolv Solution database. However, the PSR is still saved in the system. The PSR is subject to validation at the time of completion.

Note: You must be using the PSR Order Entry API for this preference to affect MetaSolv's software.

Override Default Value on PSR API Import When Label Exists on Import Structure

This system level preference is a check box that can be deselected (N) or selected (Y):

- **N:** You must populate the default values and an activity code for delete.
- **Y:** The importNGNPSR and importPSR_v3 methods overwrite the default values before making the API call when the PSROrderItemValue2 structure is populated.

You do not have to populate the value structure with the default value and an activity code for delete. You can populate it with the preferred value and an activity code of New.

Note: You must be using the PSR Order Entry API for this preference to affect MetaSolv's software.

Use Copy Item When Importing PSR Order

This system level preference is a check box that can be deselected (N) or selected (Y):

- **N:** When an item is copied during the import of a PSR order, the item, as well as all child items, are copied.
- **Y:** When an item is copied during the import of a PSR order, only the item itself is copied.

Using Metasolv Solution Inventory as the Primary Inventory for Telephone Numbers

This system level preference is a check box that can be deselected (N) or selected (Y):

- **N:** You can create a WTN while you are creating a service request.
- **Y:** You can assign the working telephone number (WTN) from the telephone number inventory to a service request, but you cannot create a WTN during service request entry.

PSRProductCatalogSession Interface Operations

Table 10–3 lists the operations in the PSRProductCatalogSession of the WDIPSR.IDL file and their accompanying notification operations.

Table 10–3 PSRProductCatalogSession Interface Operations

Operation	WDIProductCatalogNotification
exportProductCatalog_v2 - Deprecated	exportProductCatalogSucceeded_v2 exportProductCatalogFailed
exportProductCatalog_v3	exportProductCatalogSucceeded_v3 exportProductCatalogFailed
exportProductGroups	PSRExportProductGroupsSucceeded PSRExportProductGroupsFailed
exportProductCatalogWith Templates	PSRExportProductCatalogWithTemplatesSucceeded exportProductCatalogFailed

PSRProductCatalogSession Operation Descriptions

This section describes the operations defined in the WDIPSR.IDL file.

- exportProductCatalogWithTemplates

  Enables export of product catalog information for template-based products (those with an item type of System, Element, Connector, or Equipment).

PSRProvisioningSession Interface Operations

Table 10–4 lists the operations in the PSRProvisioningSession of the WDIPSR.IDL file and their accompanying notification operations.

Table 10–4 PSRProvisioningSession Interface Operations

Operation	WDIProvisioningNotification
executeFinishOrder	executeFinishOrderSucceeded executeFinishOrderFailed
executeCLSCktIDAssignment	executeCKTIDAssignmentSucceeded executeCKTIDAssignmentFailed
exportCktItems	exportCktItemsSucceeded exportCktItemsFailed
exportLSOClli	exportCktLSOCliiSucceeded exportCktLSOCliiFailed

Process flow

The section that follows contains a sample process flow for unsolicited messages. Use the sample flow as a template when you develop your own process flows.

Unsolicited Messages

When the message is initiated by the third party (unsolicited), MetaSolv Solution plays the role of the server, and the third-party application plays the role of the client. Unsolicited messages are processed asynchronously, meaning a callback mechanism is used to report back the results of an operation invoked by the third-party application.

Sample Unsolicited Process Flow for Importing a Customer

The overall process flow for importing a customer is as follows:

1. The third-party application binds to the MetaSolv Solution Application Server to get a WDIRoot object reference.
2. The third-party application invokes the *startPSRSession* operation of the WDIManager interface to get a PSRSession object reference.
3. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.
4. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference.
5. The third-party application instantiates a WDINotification object.
6. The third-party application invokes the *importNewCustomer* operation on the PSRSession interface, providing WDITransaction, WDINotification, and PSRCustomerAccount objects.
7. The MetaSolv Solution Application Server processes the invoked PSRSession operation and invokes the appropriate callback operation on the input WDINotification. In this example, the operations are *PSRCustomerExportSucceeded* or *PSRExportFailed* for exporting, and *PSRImportSucceeded* or *PSRImportFailed* for imports.
8. If the *PSRImportSucceeded* operation is invoked, the third-party application invokes the *commit* operation of the WDITransaction interface. If the *PSRExportFailed* operation is invoked, a WDIError sequence describing the error is returned to the third-party application. The third-party application then performs the appropriate

error handling routine. In the case of an import failing, the third-party application should rollback the transaction.

9. The third-party application invokes the *destroyPSRSession* operation of the WDIManager interface.
10. The third-party application invokes the *destroyTransaction* operation on the WDIManager interface.
11. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

Import Notifications

When the import of a new object succeeds, the document number is populated with the ID of the new record. For PSRCustomerAccount imports, it is custAcctID. For PSRServiceOrder imports, it is documentNumber. For PSRSvcLocation imports, it is endUserLocationID, endUserLocationType is whatever was provided during the import.

The documentNumber field will never pass back a value other than 0 on any of the failed notification operations. The success notification operations will return a value of 0, except in the following situations:

- An existing PSR API operation has passed any unique integer value that would allow the client to uniquely identify the success notification operation. This is usually an API operation method using a specific customer account ID or order (document) number to export/import data.
- A new (not versioned _v2 or _v3) operation is added to the PSR API which can correctly implement the field for the client to send the documentNumber to the server.

PSR API Date Handling

To indicate that a date should be considered null, send "0" for the day, "0" for the month, and "0" for the year. If you supply a year that is fewer than four digits, 1900 is added to the value to determine the year. If four digits are provided, it is assumed that this is the exact year.

Table 10–5 provides information on how the dates that you specify are interpreted.

Table 10–5 Specified and Interpreted Dates

Specified Date	Interpreted Date
1/1/99	January 1, 1999
1/1/101	January 1, 2001
1/1/1	January 1, 1901
1/1/2001	January 1, 2001

Batch Operations in PSR API Exports

The failure of one item in an export of multiple items does not cause the failure of the entire export. Instead, the items that succeed are returned, as well as a sequence of error messages describing the failed operations. When the client requests an operation that can result in multiple items being returned, both the exportSucceeded and exportFailed operations can be called. The exportSucceeded operation is always called, even if all items failed export, in which case the returned sequence is empty. Although

the exportFailed operation is generally called first, it cannot be guaranteed in multi-threaded environments.

The export of items such as Telephone Number Inventory Export result in a large amount of data, which can be difficult to manage. Therefore, this operation was broken into a series of exports. Groups of 100 telephone numbers at a time, minus failed retrievals, are sent to the notification object with the PSRTelNbrBatchSucceeded operation. After all such operations have finished, the notification object is sent with the PSRTelNbrExportComplete operation.

Export Search Criteria

The PSR API provides a means to export data based on specific search criteria. You can perform the search using the specifications identified in the SearchableField enumerated type in the WDIPSR.IDL file. Additionally, there are specific operations available to specify the criteria, such as EQUAL, LIKE, and so on. These are defined in the SearchOperation enumerated type.

Finally, the SearchCriteria struct contains one SearchableField reference, one SearchOperation reference, and a string value to search for. Note that all fields that can be searched are not necessarily strings; in the cases of numeric values, a string representation of the value is expected. In the case of enumerated values, a string representation of the numeric value of the enumerated type is expected. It is possible to provide multiple SearchCriteria for an operation, which means that the resulting values must meet all of the specified criteria (an AND operation).

MetaSolv Solution Product Specification and Product Catalog

This section provides information on MetaSolv Solution product specification and product catalog.

Products

MetaSolv Solution has the following product levels:

- Item Types
- Product Specifications
- Product Catalog

Item Types

Think of item types as the MetaSolv Solution rules. These are items and relationships that MetaSolv Solution has predefined for products and services. Examples include line products, which can have attributes such as lines, system options, and features. These attributes allow the MetaSolv Solution code to determine what kind of service a product is and ensure certain data is collected and specific processing occurs. For example, circuits are built (in the background) for dial tone lines so they can be designed later.

Product Specifications

Product specifications are engineering rules defined by technical staff in conjunction with product management. These specifications are supported by engineers and network designers for specific implementations. Using the item type examples above (line products, with attributes of lines, system options, and features), a customer could create a product specification for dial tone (POTS) service as a line product with lines

that have a system option of Hunt Groups. Each line may have several features such as call waiting, call forwarding, caller identification, three way conferencing, and so on. These specifications are building blocks that have pre-defined relationships designed so that a product catalog can be built using these building blocks. People creating the product catalog need not be worried about the provisioning/designing aspects of a product.

Product Catalog

The product catalog is the marketing rules defined by product management. The product catalog addresses three primary marketing issues: 1) Product availability (by location or by business/market segment), 2) Pricing and 3) Packaging. Again, continuing our earlier example, a residential dial tone product, a basic business dial tone product and an enhanced business dial tone product could all be built from the product specification example. Each product may have different pricing, market segments, etc. Multiple product catalogs can be built from one product specification.

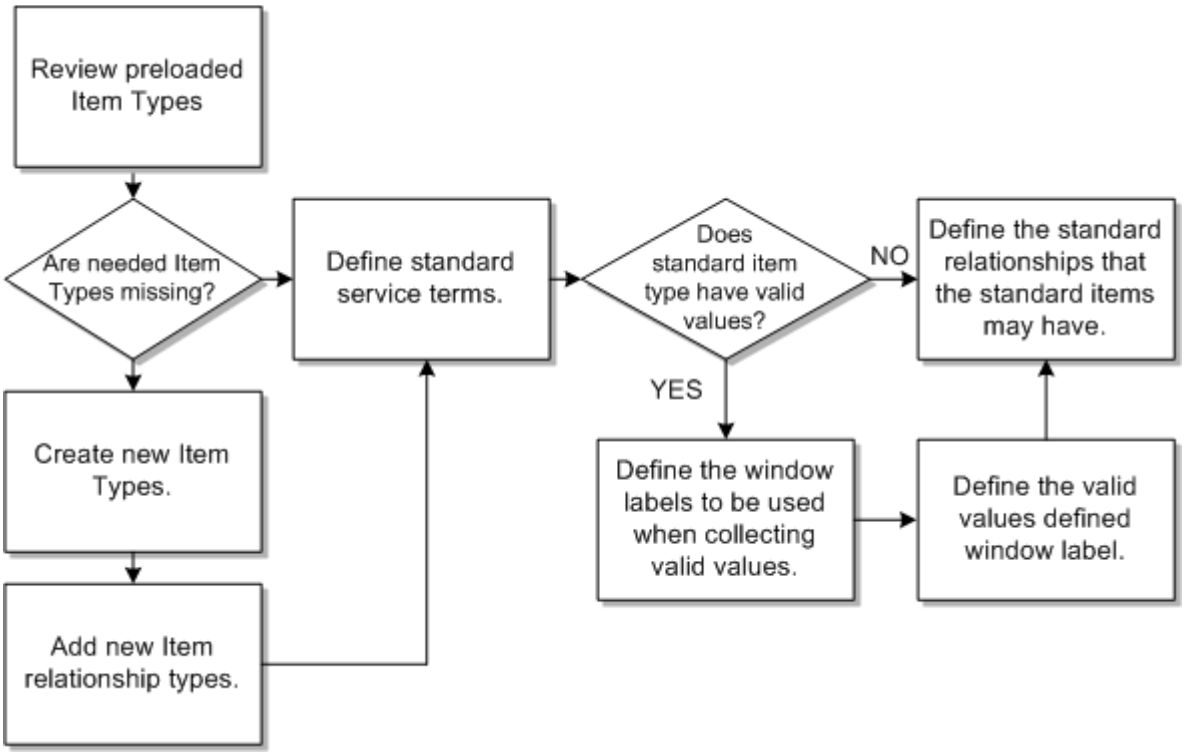
More About Products

An item is the generic name for products and options. Setting up products and options is a two-step process. First, the product specifications and the rules for relationships between product specifications must be defined. The next step is creating the product catalog and designing how the items will be sold. The product catalogs must follow the rules set up in the first step. When setting up product catalogs, the marketing item is the level one item. Usually products are the level one items in the product catalog. The highest level in the hierarchy of items is defined as a level one item.

More About Product Specifications

Figure 10–2 describes the basic process flow for setting up product specifications.

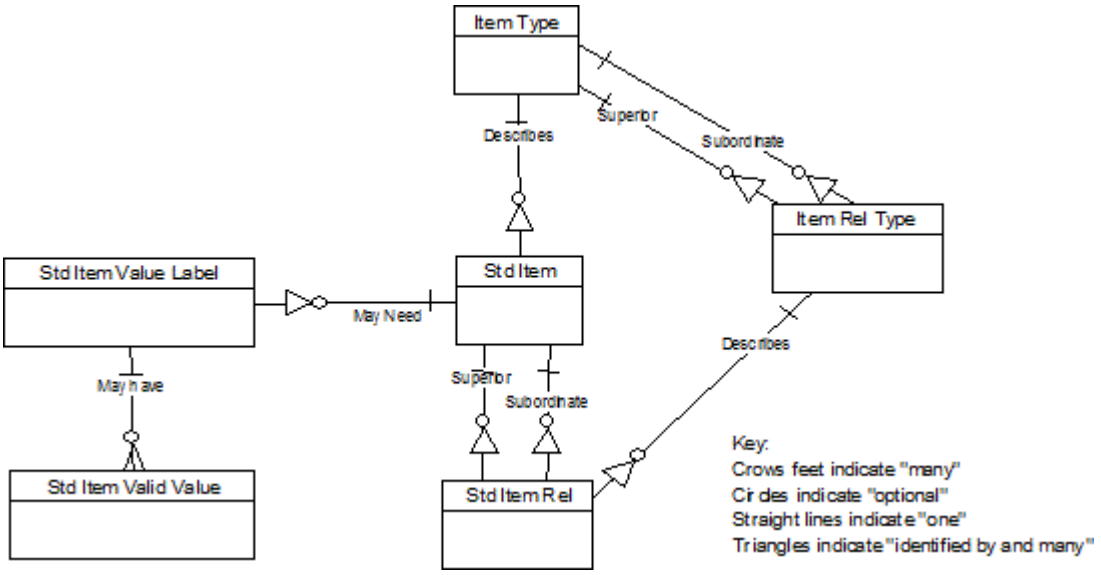
Figure 10–2 Standard Item Setup



Note: All references to standard item in this flow refer to product specifications. References to item types refer to MetaSolv Solution-defined item types.

Figure 10–3 shows the standard item setup data model.

Figure 10–3 Standard Item Setup Data Model



Note: All entities with standard_item in the name hold the product specification information. Entities with item_type in the name store MetaSolv Solution-defined items.

The item type and item relationship type entities are preloaded with MetaSolv Solution. These two entities define the common characteristics that are handled in the application. An item type and its characteristics are pre-defined. Characteristics might include an item category (such as product, option, line, trunk, etc.), whether the item type must have a premise, and the item type's processing path. (The processing path defines the processing that must take place for an item type.)

Table 10–6 describes the item types's processing path.

Table 10–6 Item Type Processing Path

Processing Path	Acronym	Description
Local Telephone Line Service	LTLS	The item will have a premise and will have lines.
Local Telephone Trunk Service	LTTS	The item will have a premise and trunks.
Non-Premise Service	NPS	Indicates the item will not have a service location and the billing address will be used for the service location.
Non Switched Services	NSS	Product catalogs that fall in this path will require circuit assignments and two locations.

For instance, a type of item might be "line product," which would be characterized as a product. Another type might be "line", which would be characterized as a physical item (meaning a circuit will be created for design purposes) with a premise. The relationship between these two item types would be defined in the item type relationship table with line product being the superior item and line being the subordinate item.

When users define the product specifications, they delineate the actual items that can be sold. For instance, a product specification of basic business line might be set up with a type of line product. Some rules of use are defined here. For example, if you want a disconnect reason to be entered when this item is disconnected, you could set that here. Line types are another product specification that can be set up with a basic business line. The rules that can be defined for this item include the disconnect reason must be entered, and when the item is ordered the question "how many do you want?" must be asked.

Table 10–7 lists the characteristics that you must define if you use the Arbor/BP billing system or flow through provisioning with PSR.

Table 10–7 Standard Items

Table	Column	Description
standard Item	Arbor EMF Ind (called Service Instance on the window)	Indicates that the item translates to a service instance in the billing system. These columns may be used for any billing interface, not just the Kenan Arbor billing system.
standard Item	Arbor Usage Guiding Key (called Guiding Key on the window)	If the Arbor EMF Ind is set on, a usage guiding key is necessary. These columns may be used for any billing interface, not just the Kenan Arbor billing system.

Table 10–7 (Cont.) Standard Items

Table	Column	Description
standard Item	Flow Through Provisioning Nm	The command recognized by the FTP Gateway for a PSR item. If this column has a value, it can be used in the flow-through provisioning process.
standard Item	Switch Provisioning Ind	Indicates that the product specification will be used for switch provisioning. In an installation where the FTP Gateway is used this indicator directs the item to be used in the flow through provisioning process. If the FTP Gateway is not used, this indicator directs the item to be part of a report to be used to perform the switch provisioning.
standard Item	Circuit Design Ind	Indicates that the product specification will be used in the circuit design process.

Note: The standard item table is referring to product specifications

Next the standard item relationship must be defined. If basic business is set up as the highest item in the hierarchy (a product), it would be defined in this table with basic business having no superior item. Basic business line would be set up as a subordinate to basic business. Valid values may also be specified during the product specification setup process. If you have an item that requires one or more values to be collected, a name (label) is setup for each value collected and a list or range of valid values defined. A default value may also be defined. An example illustrating this feature might be an item called Start Type. A product specification of Start Type would have a label of "Type" and a value list of "ground" or "loop."

More About Product Catalogs

Figure 10–4 describes the basic process flow for setting up product catalogs (things to sell). All references to specifications in this flow refer to product catalogs. References to items refer to product specifications.

Figure 10–4 Specification Setup

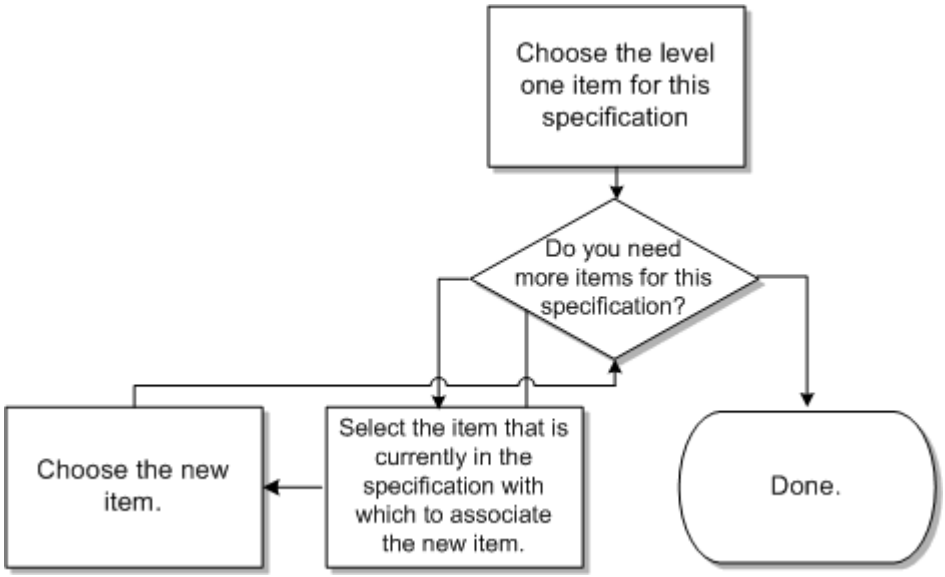
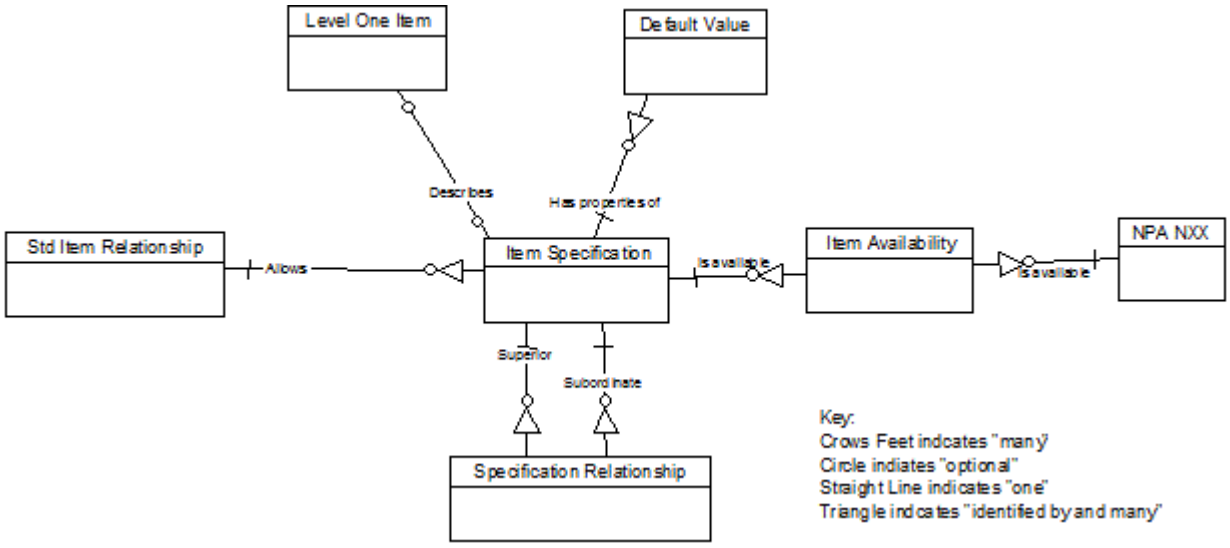


Figure 10–5 shows the item specification data model.

Figure 10–5 Item Specification Data Model



Note: Entities with specification in the name store the product catalog information.

When creating a product catalog for a product, choose the product you want to set up from a list of standard items. The relationship (product specifications) of these items must have no superior items, for example, basic business. Once you have chosen the level one item, the next level of items may be selected from a list of standard items whose relationship has basic business as the superior item. In our example, basic business lines may be selected. This process sets up the product catalog relationship. For level one items there is other information that needs to be collected such as type of service, offering type (wholesale, resell, and retail), taxing information, and tariff information. A product catalog name may be entered. Other information may be specified about any item including information describing the availability of the item (by Network Area), the from and to effective dates, and whether or not it is required or standard. Pricing information may also be entered at this time (see the pricing area of this document for specifics).

Packages

Defining a package is the same as defining a product. Currently, MetaSolv Solution allows packages to be set up within a product for example packages of features for a line. MetaSolv Solution does not currently support packages across products (for example, packaging lines and trunk groups together).

11

The Switch Provisioning Activation API

The Switch Provisioning Activation API provides the IDL for switch provisioning activation. The Switch Provisioning Activation API retrieves Design Layout Records (DLRs), switch translation, and flow-through information for a given WDIEvent.

Your application is responsible for managing all database transactions, including *commit* and *rollback* processing.

Functionality

The Switch Provisioning Activation API facilitates flow-through provisioning for switched orders initiated from the PSR module of the Order Management subsystem, and enables flow-through provisioning of dialtone orders. The Switch Provisioning Activation API is directly involved with Oracle Communications MetaSolv Solution and invokes the same rules.

Essential Terminology

Table 11–1 lists the terms that identify information and concepts that are required to understand the flow-through provisioning using the APIs.

Table 11–1 Switch Provisioning Terminology

Term	Definition
Activation product	A network management system (NMS), such as Lucent Technology's ACTIVEVIEW product line.
Activation server	An application developed by you or a third party that integrates with MetaSolv Solution to export provisioning data and communicate the data to one or more activation products.

Switch Provisioning Activation Interface

This section provides information about the Switch Provisioning Activation interface.

DLRSession Interfaces

Table 11–2 lists the operations available in the DLRSession of the **WDIDLR.IDL** file that is used by the Switch Provisioning Activation API.

Table 11–2 DLRSession Interface Operations

Operations	WDINotification
getSwitchActivation_v2	switchActivationGetSucceeded_v2 switchActivationGetFailed
getSwitchActivation_v4	SwitchActivationGetSucceeded_v4 SwitchActivationGetFailed_v4
getSwitchActivation_v5	SwitchActivationGetSucceeded_v5 SwitchActivationGetFailed_v5

DLRSession Interface Operations

The following list contains the DLRSession operations of the WDIDLR.IDL file that relate to switch provisioning activation:

- getSwitchActivation_v2, get SwitchActivation_v4, and get switch Activation_v5

Process Flows

This section contains sample process flows for each type of message: solicited and unsolicited. Use the sample flow as a template for developing your own process flows.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. MetaSolv Solution plays the role of the client, and the third-party activation server plays the role of the server.

Table 11–3 lists the interfaces and operations that the third-party application implements using the IDL file provided with the DLR API.

Table 11–3 Switch Provisioning API Interfaces Solicited Messages Operations

Interface	For Implementing These Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	N/A
WDISignal	eventOccurred eventTerminated
WDIInSignal	N/A

Sample Solicited Message Process Flow

When MetaSolv Solution is the client, the overall process flows as follows:

1. The API client binds to the third-party server to get a WDIRoot object reference.
2. The API client invokes the *connect* operation of the WDIRoot interface, and the *connect* operation yields a WDIManager object reference.
3. The API client invokes the *startSignal* operation of the WDIManager interface to get a WDISignal object reference.

4. The API client invokes the *eventOccurred* operation of the WDISignal interface to notify the third-party application that an event registered to them has occurred within MetaSolv Solution.
5. The API client invokes the *destroySignal* operation of the WDIManager interface.
6. The API client invokes the *disconnect* operation of the WDIRoot interface.
7. Once the third-party server completes processing, possibly involving additional unsolicited messages to MetaSolv Solution, the third party performs a bind to the MetaSolv Solution Application Server and follows the same process described above for the client with the exception that the *eventCompleted/Errored* operations are invoked passing the original WDIEvent structure.

If the third-party application encounters an error, it throws a WDIExcp as defined by the IDL. The client handles CORBA system exceptions and WDIExcp exceptions.

Unsolicited Messages

An unsolicited message is a message initiated by the third-party application. MetaSolv Solution plays the role of the server and a third-party application plays the role of the client with the exception of the callback processing.

Table 11–4 lists the interfaces and operations that MetaSolv Solution implements using the IDL file provided with the Switch Provisioning Activation API.

Table 11–4 Switch Provisioning Interfaces Unsolicited Messages Operations

Interface	For Implementing These Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	commit rollback
DLRSession	getSwitchActivation_v5

Table 11–5 lists the interfaces and operations for which the third-party application is responsible.

Table 11–5 Switch Provisioning Third-party Application Interfaces and Operations

Interface	For Implementing These Operations
WDINotification	switchActivationGetSucceeded_v5 switchActivationGetFailed_v5

Process Flow for Exporting Switch Provisioning Activation Information

The overall process flow for exporting a DLR follows:

1. The third-party application binds to the MetaSolv Solution Application Server to get a WDIRoot object reference.
2. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.

3. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference and start a database transaction.
4. The third-party application invokes the *startDLRSession* operation of the WDIManager interface to get a DLRSession object reference.
5. The third-party application instantiates a third-party implementation of a WDINotification object.
6. The third-party application invokes the *getSwitchActivation* operation of the DLRSession object, passing the WDINotification object.
7. The SwitchActivation data structure is returned asynchronously through invocation of the *switchActivationGetSucceeded/Failed)* operation of the WDINotification object.
8. The third-party application invokes the *destroyDLRSession* operation of the WDIManager interface.
9. The third-party application invokes the *destroyTransaction* operation of the WDIManager interface.
10. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

Implementation Concepts

This section describes the issues that you must be familiar with when building a mediation server application for flow-through provisioning.

What Are Network Nodes and Network Node Types?

Network nodes are the equipment that manages the circuits in the network. They are identified by a unique target identifier (TID). TIDs are used to search for devices on the network. Commands are sent to the network node for flow-through provisioning. For example, a user might designate one network node as the host network element that communicates with the network management system. Essentially, a network node is any device that can be provisioned through software. Network nodes can contain one or more pieces of equipment, and can be directly associated with flow-through provisioning plans on the Network Node Type window in the Infrastructure module.

If flow-through plans are used, the flow-through provisioning process cannot occur without network nodes. Network node types are used in the flow-through provisioning process to categorize network nodes into groups. Network node types represent the activation vendor's requirements for activating the network element, and they are used in the flow-through provisioning process to limit the number of flow-through provisioning plans required.

What are Flow-through Provisioning Plans and Commands?

Flow-through provisioning plans and flow-through provisioning commands are MetaSolv Solution concepts that define optional additional parameters used in the flow-through provisioning process that are not a part of MetaSolv Solution. Below are examples of some of the types of flow-through provisioning plans and commands that can be created:

- Plan
  - Activate a DACS

- Command
  - Config Port A
  - Config Port B
- Parameters
  - Direction: 1 way
  - Direction: 2 way
  - Alarming

The number of flow-through provisioning plans, commands, and parameters that are created will vary according to the requirements of the activation product used for the flow-through provisioning process. The nature of flow-through provisioning plans and commands is to allow MetaSolv Solution to work with any selected activation product. That is, MetaSolv Solution only captures TID, port addresses, and cross-connects for flow-through provisioning. Flow-through provisioning plans and commands provide the ability to capture all the information the activation vendor requires.

Note: If the selected activation product only requires the defaults for flow-through provisioning, then it is not necessary to use the PSR module in the flow-through provisioning process at all.

What Are Design Layout Records (DLRs)?

A design layout record (DLR) is a document that contains the technical information that describes the physical layout of a circuit at a given location.

What are Tech Translation Sheets?

The tech translation sheet defines the items required to provision the service in the switch. For switch provisioning activation, once the order is entered, the product and options ordered are the basis for the tech translation sheet.

What are Virtual Layout Records (VLRs)?

A virtual layout record (VLR) is a MetaSolv Solution-defined document that contains the technical information that describes the layout of the physical components of an ATM or Frame Relay virtual circuit, and the relationship of the physical components to the logical components (the cloud) of that circuit.

Software Modules and Subsystems Used in Flow-through Provisioning

The Switched Provisioning Activation and Transport Provisioning Activation APIs use the following modules and subsystems in the to complete the flow-through provisioning process:

- Equipment Administration module
- Infrastructure module
- Product Service Request (PSR) module
- Service Provisioning subsystem
- Work Management subsystem

Equipment Administration Module

The flow-through provisioning process uses the Equipment Administration module to define the following:

- Target identifier (TID) that the activation vendor recognizes
- Network node with which the equipment is associated

> **Note:** A network node must be associated with every piece of equipment (or at a higher level in the equipment hierarchy) used for flow-through provisioning. A network node type must be associated with every network node used for flow-through provisioning.

Infrastructure Module

The Infrastructure module is used in the flow-through provisioning process to:

- Define new network node types
- Associate network node types with flow-through provisioning plans and rate codes

Additionally, the user can access the PSR module's Product Catalog function through the Infrastructure module. The user will only use the Infrastructure module for flow-through provisioning if they need to specify additional data for a network node.

Product Service Request Module

The Product Service Request (PSR) module is used in the flow-through provisioning process to:

- Set up flow-through provisioning plans and commands
- Enter a service request
- Provide service request information related to flow-through provisioning on the tech translation sheet for switch translations

More specifically, the PSR module's Product Catalog function is used in the flow-through provisioning process to define the features that appear on the PSR, as well as the options on those features. Options on the service request used for the flow-through provisioning process include flow-through provisioning plans and commands. These options often have default values, and when a PSR is entered with these options, the service request includes the required default values (also referred to as parameters). The MetaSolv Solution user can use the Product Specifications window to determine if the values or parameters appear on the tech translation sheet.

> **Note:** All parameters necessary to the flow-through provisioning process (except equipment parameters) are defined in the Product Catalog function.

Service Provisioning Subsystem

The flow-through provisioning process uses the Service Provisioning subsystem to:

- Design the circuit(s) on the PSR used for flow-through provisioning
- Verify and modify the flow-through provisioning parameters that are set up in the PSR module

The Service Provisioning subsystem also provides the flow-through provisioning information (such as network node type and network node address) that appears on the CLR, DLR, VLR, and tech translation sheet.

Work Management Subsystem

The Work Management subsystem is used in the flow-through provisioning process to:

- Associate a gateway event with a provisioning plan task
- Initiate a gateway event
- Verify the gateway event is complete

Gateway events define when MetaSolv Solution should send flow-through provisioning information to the activation application for processing.

Flow-through Provisioning Process

The flow-through provisioning process is used in MetaSolv Solution to:

- Order and provision services associated with the line side of a switch
- Engineer a service request and provision it without re-entering activation information

> **Note:** Line side activation includes provisioning dialtone services through a switch. The activation process occurs outside of MetaSolv Solution.

See the online Help for detailed instructions on using the flow-through provisioning process.

Signal Handler

The signal handler module implements the interfaces required to handle standard gateway events from MetaSolv Solution clients. This module is also responsible for updating gateway event status to “In Progress”.

The outbound signals sent by the client to your activation server are the flow-through provisioning gateway events. These events are defined at the service item level. Each service item (for example, a phone line, a WATS line, or an ATM/Frame circuit) on the order will have the flow-through provisioning gateway event associated with it. As a result, when an order is processed by the Work Management subsystem, your activation server can potentially receive as many gateway events as there are service items in the order. For example, if a transport provisioning order for ASR equipment comprises six special access circuits, your activation server receives six separate gateway events from the client.

Each gateway event associated with a service item in a service request can be processed independently of the gateway events for any other service item.

Ensure that the implementation conforms to the pattern described in "Outbound Signals – Gateway Events". The signal handler module should implement a WDIDLR module with all the interfaces and operations specified in Table 2–6, " Outbound Gateway Event Operations Required For All APIs". Event status updates are performed through DLRSERVER.

Upon receiving an outbound signal conveying gateway event information from the client, the signal handler module activates the request handler module and hands off

the event information that was received. In order to avoid locking up the client, it is recommended that the signal handler should return control to the client immediately upon activating the request handler module and updating the event status.

Request Handler

The request handler module retrieves activation data from the MetaSolv Solution database by invoking the operation *getSwitchActivation* on DLRSERVER to retrieve switch activation data.

The operation is a standard data export operation that conforms to "Asynchronous Interaction Pattern". This provisioning operation accepts a WDIEvent parameter. This allows the request handler to retrieve provisioning data from the database in a single step.The request handler passes the gateway event structure that was received from the client, and DLRSERVER retrieves the required provisioning data.

It is important to understand the data types that are involved in the two operations listed above. Data type definitions can be found in file WDIDLRTYPES_v5.IDL. The following Switch Activation data structure is returned to the caller (through callback invocation):

Example 11–1 Switch Provisioning Data Structure Example

```
struct SwitchActivation {
    DLR  dlr;
    DLRSwitchTranslation        switchtranslation;
    ActivationCommandPlanSeq    activationCommandPlans;
};
```

The ActivationCommandPlanSeq data type delivers the FTP Plan for this service item.

Formatting/Translation Module

The formatting/translation module handles two-way data translation and format conversion required for communicating with the activation product. This module's services are used by the other modules.

Response Handler

The response handler module handles responses received from the activation product. It performs the necessary reverse translation/formatting using the formatting/translation module and then determines the operation status. Based on the success or failure determination, this module updates gateway event status to “Completed” or “Errored”. Design of this module depends upon factors such as the synchronous/asynchronous and online/batch nature of the interaction with activation product.

Date/Time Format

Dates are returned using the MetaSolv:CORBA:WDIUtil:MSVDate structure, which stores the date and time information as a string of the form YYYYMMDDHHMMSS.

CORBA Substructures

The CORBA specification does not allow uninitialized values for structures or types embedded within other structures. In the case of no data, a sequence of length "0" is returned.

Design Considerations

To obtain the full benefit of the automated flow-through capabilities of these APIs, gateway events must be associated with tasks in Work Management provisioning plans. MetaSolv Solution is pre-configured with gateway templates and gateway event templates for Switch Provisioning.

See the online Help for detailed instructions on using the flow-through provisioning process.

In order to ensure that the provisioning information provided by the Switched Provisioning Activation API is sufficiently completed to be used by your network management system, care must be used when ordering the service.

The PSR module captures default values for items that have pick lists. With flow-through provisioning, defaults are also needed for editable fields. If defaults are not provided, a user would be required to manually enter the same value on every line for an order. Providing a default value in the product catalog for product specifications that are required for flow-through provisioning streamlines the ordering process.

For transport provisioning activation, the network node target identifier (TID) and the equipment port address assignment identifier (AID) in the database identify the equipment and port address. These items should either be used directly by your application or your application should maintain a cross-reference between the identifiers used by your application and the MetaSolv Solution-supplied TID and AID.

Just as the provisioning of switch features requires additional parameters, the provisioning of transport equipment requires additional parameters as well. The transport equipment for dialtone lines is usually digital loop carrier. The MetaSolv Solution CLR represents the provisioning information for this type of equipment. The CLR captures the TID and the AID for the DLC equipment, which is part of the information that is required for activating the service. The TID is determined by identifying the Network Node to which the equipment belongs. The AID is determined using the assignment information that is gathered on the CLR. To provision transport equipment, additional parameters are usually required. These parameters will vary by type of equipment, by transmission rate, and by activation vendor.

12

The Transport Provisioning Activation API

The Transport Provisioning Activation API supports flow-through provisioning of different kinds of circuit designs. This API enables third-party network management systems to export provisioning information from the Oracle Communications MetaSolv Solution database and use that information to physically implement the design.

The Transport Provisioning Activation API:

- Provides a vendor-independent interface to enable flow-through provisioning of Frame Relay and ATM circuits.
- Provides flow-through information about any transport equipment assigned to a DLR (for example: SONET and DACS).
- Exposes the VLR through an API so customers can write Web applications that display the VLR through a thin client.

Functionality

The Transport Provisioning Activation API provides the IDL for retrieving DLR, VLR and flow-through information for a given WDIEvent. If the value for the returned "Type" data element is V, VLR information exists for the circuit; otherwise DLR information exists.

The third-party application is responsible for managing all database transactions, including commit and rollback processing.

Essential Terminology

Table 12–1 defines the terms that identify the concepts and information that are required to understand flow-through provisioning using the APIs.

Table 12–1 Transport Provisioning API Terminology

Term	Definition
Activation product	A network management system (NMS), such as Lucent Technology's ACTIVEVIEW product line.
Activation server	An application developed by you or a third party that integrates with MetaSolv Solution to export provisioning data and communicate the data to one or more activation products.

Transport Provisioning Activation Interface

This section provides information about the Transport Activation interface.

DLRSession Interfaces

Table 12–2 lists the operations available in the DLRSession of the **WDIDLR.IDL** file that are used by the Transport Provisioning Activation API.

Table 12–2 DLR Session WDINotification Operations

Operations	WDINotification
getTransportProvisioning_v2	transportProvisioningGetSucceeded_v2 transportProvisioningGetFailed
getTransportProvisioning_v4	transportProvisioningGetSucceeded_v4 transportProvisioningGetFailed_v4
getTransportProvisioning_v5	transportProvisioningGetSucceeded_v5 transportProvisioningGetFailed_5

DLRSession Interface Operation

The following list contains the operation used in the DLRSession of the WDIDLR.IDL file:

- getTransportProvisioning_v2,getTransportProvisioning_v4, and getTransportProvisionin_v5
- getVLR_v2

  This operation replaces the getVLR operation from earlier releases.

Process Flows

This section contains sample process flows for each type of signal: solicited and unsolicited. Use the sample flow as a template for developing your own process flows.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. With this scenario, MetaSolv Solution plays the role of the client, and the third-party activation server plays the role of the server.

Table 12–3 lists the interfaces and operations that the third-party application implements using the IDL file provided with the DLR API.

Table 12–3 Transport Provisioning API Interfaces Solicited Messages Operations

Interface	Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	N/A

Table 12–3 (Cont.) Transport Provisioning API Interfaces Solicited Messages Operations

Interface	Operations
WDISignal	eventOccurred eventTerminated
WDIInSignal	N/A

Sample Solicited Message Process Flow

When MetaSolv Solution is the client, the overall process flows as follows:

1. The API client binds to the third-party server to get a WDIRoot object reference.
2. The API client invokes the *connect* operation of the WDIRoot interface, and the *connect* operation yields a WDIManager object reference.
3. The API client invokes the *startSignal* operation of the WDIManager interface to get a WDISignal object reference.
4. The API client invokes the *eventOccurred* operation of the WDISignal interface passing a WDIEvent structure to notify the third-party vendor that an event registered to them has occurred within MetaSolv Solution.
5. The API client invokes the *destroySignal* operation of the WDIManager interface.
6. The API client invokes the *disconnect* operation of the WDIRoot interface.
7. Once the third-party server completes processing, possibly involving additional unsolicited messages to the MetaSolv Solution Application Server, the third party binds to the application server and follows the same process described above for the MetaSolv Solution client with the exception that the *eventCompleted/Errored* operations are invoked passing the original WDIEvent structure.

If the third-party application encounters an error, it throws a WDIExcp as defined by the IDL. The client handles CORBA system exceptions and WDIExcp exceptions.

Unsolicited Messages

An unsolicited message is a message initiated by the third-party software. MetaSolv Solution plays the role of the server, and a third-party application plays the role of the client with the exception of the callback processing.

Table 12–4 lists the interfaces and operations that MetaSolv Solution implements using the IDL files provided with the Transport Provisioning Activation API.

Table 12–4 Transport Provisioning API Interfaces Unsolicited Messages Operations

Interface	Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	commit rollback
DLRSession	getTransportProvisioning

Table 12–5 lists the interfaces and operations for which the third-party application is responsible.

Table 12–5 Transport Provisioning API Third-party Interfaces and Operations

Interface	For Implementing These Operations
WDINotification	transportProvisioningGetSucceeded_v5 transportProvisioningGetFailed_v5

Sample Unsolicited Message Process Flow for Exporting Transport Provisioning Activation Information

The overall process flow for exporting Transport Provisioning Activation is as follows:

1. The third-party application binds to the MetaSolv Solution Application Server to get a WDIRoot object reference.
2. The third-party application invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.
3. The third-party application invokes the *startTransaction* operation of the WDIRoot interface to get a WDITransaction object reference and starts a database transaction.
4. The third-party application invokes the *startDLRSession* operation of the WDIManager interface to get a DLRSession object reference.
5. The third-party application instantiates a third-party implementation of a WDINotification object.
6. The third-party application invokes the *getTransportProvisioning* operation of the DLRSession object, passing the WDINotification object.
7. The TransportProvisioning data structure is returned asynchronously through invocation of the *transportProvisioningGetSucceeded/Failed* operation of the WDINotification object.
8. The third-party application invokes the *destroyDLRSession* operation of the WDIManager interface.
9. The third-party application invokes the destroyTransaction operation of the WDIManager interface.
10. The third-party application invokes the *disconnect* operation of the WDIRoot interface.

Implementation Concepts

This section describes the issues that you must be familiar with when building a mediation server application for flow-through provisioning.

What are Network Nodes and Network Node Types?

Network nodes are the equipment that manages the circuits in the network. They are identified by a unique target identifier (TID). TIDs are used to search for devices on the network. Commands are sent to the network node for flow-through provisioning. For example, a user might designate one network node as the host network element that communicates with the network management system. Essentially, a network node is any device that can be provisioned through software. Network nodes can contain

one or more pieces of equipment, and can be directly associated with flow-through provisioning plans on the Network Node Type window in the Infrastructure module.

If flow-through plans are used, the flow-through provisioning process cannot occur without network nodes. Network node types are used in the flow-through provisioning process to categorize network nodes into groups. Network node types represent the activation vendor's requirements for activating the network element, and they are used in the flow-through provisioning process to limit the number of flow-through provisioning plans required.

What are Flow-through Provisioning Plans and Commands?

Flow-through provisioning plans and flow-through provisioning commands are MetaSolv Solution concepts that define optional additional parameters used in the flow-through provisioning process that are not a part of MetaSolv Solution. Below are examples of some of the types of flow-through provisioning plans and commands that can be created:

- Plan
  - Activate a DACS
- Command
  - Config Port A
  - Config Port B
- Parameters
  - Direction: 1 way
  - Direction: 2 way
  - Alarming

The number of flow-through provisioning plans, commands, and parameters that are created will vary according to the requirements of the activation product used for the flow-through provisioning process. The nature of flow-through provisioning plans and commands is to allow MetaSolv Solution to work with any selected activation product. That is, MetaSolv Solution only captures TID, port addresses, and cross-connects for flow-through provisioning. Flow-through provisioning plans and commands provide the ability to capture all the information the activation vendor requires.

Note: If the selected activation product only requires what the defaults for flow-through provisioning, then it is not necessary to use the PSR module in the flow-through provisioning process at all.

What Are Design Layout Records (DLRs)?

A design layout record (DLR) is a document that contains the technical information that describes the physical layout of a circuit at a given location.

What Are Tech Translation Sheets?

The tech translation sheet defines the items required to provision the service in the switch. For switch provisioning activation, once the order is entered, the product and options ordered are the basis for the tech translation sheet.

What Are Virtual Layout Records (VLRs)?

A virtual layout record (VLR) is a MetaSolv Solution-defined document that contains the technical information that describes the layout of the physical components of an ATM or Frame Relay virtual circuit, and the relationship of the physical components to the logical components (the cloud) of that circuit.

Software Modules and Subsystems Used in Flow-through Provisioning

The Transport Provisioning Activation API uses the following modules and subsystems to complete the flow-through provisioning process:

- Equipment Administration module
- Infrastructure module
- Product Service Request (PSR) module
- Service Provisioning subsystem
- Work Management subsystem

Equipment Administration Module

The flow-through provisioning process uses the Equipment Administration module to define the following:

- Target identifier (TID) that the activation vendor recognizes
- Network node with which the equipment is associated

> **Note:** A network node must be associated with every piece of equipment (or at a higher level in the equipment hierarchy) used for flow-through provisioning. A network node type must be associated with every network node used for flow-through provisioning.

Infrastructure Module

The Infrastructure module is used in the flow-through provisioning process to:

- Define new network node types
- Associate network node types with flow-through provisioning plans and rate codes

Additionally, the user can access the PSR module's Product Catalog function through the Infrastructure module. The user will only use the Infrastructure module for flow-through provisioning if they need to specify additional data for a network node.

Product Service Request Module

The Product Service Request (PSR) module is used in the flow-through provisioning process to:

- Set up flow-through provisioning plans and commands
- Enter a service request
- Provide service request information related to flow-through provisioning on the tech translation sheet for switch translations

More specifically, the PSR module's Product Catalog function is used in the flow-through provisioning process to define the features that appear on the PSR, as

well as the options on those features. Options on the service request used for the flow-through provisioning process include flow-through provisioning plans and commands. These options often have default values, and when a PSR is entered with these options, the service request includes the required default values (also referred to as parameters). The user can use the Product Specifications window to determine if the values or parameters appear on the tech translation sheet.

> **Note:** All parameters necessary to the flow-through provisioning process (except equipment parameters) are defined in the Product Catalog function.

Service Provisioning Subsystem

The flow-through provisioning process uses the Service Provisioning subsystem to:

- Design the circuit(s) on the PSR used for flow-through provisioning
- Verify and modify the flow-through provisioning parameters that are set up in the PSR module

The Service Provisioning subsystem also provides the flow-through provisioning information (such as network node type and network node address) that appears on the CLR, DLR, VLR, and tech translation sheet.

Work Management Subsystem

The Work Management subsystem is used in the flow-through provisioning process to:

- Associate a gateway event with a provisioning plan task
- Initiate a gateway event
- Verify the gateway event is complete

Gateway events define when MetaSolv Solution should send flow-through provisioning information to the activation application for processing.

Flow-through Provisioning Process

The flow-through provisioning process is used in the MetaSolv Solution client software to:

- Order and provision services associated with the line side of a switch
- Engineer a service request and provision it without re-entering activation information

> **Note:** Line side activation includes provisioning dialtone services through a switch. The activation process occurs outside of MetaSolv Solution.

See the online Help for detailed instructions on using the flow-through provisioning process.

For flow-through provisioning, the Transport Provisioning API calculates and provides the physical/logical port values for both the A location of the equipment and the Z location of the equipment.

The Transport Provisioning API provides the API user with physical/logical port values and a list of circuit positions ridden for each bandwidth circuit that supports the PVC.

When multiple circuit positions are ridden by a bandwidth circuit, the API throws an exception if all of the circuit positions are not associated with the same logical port.

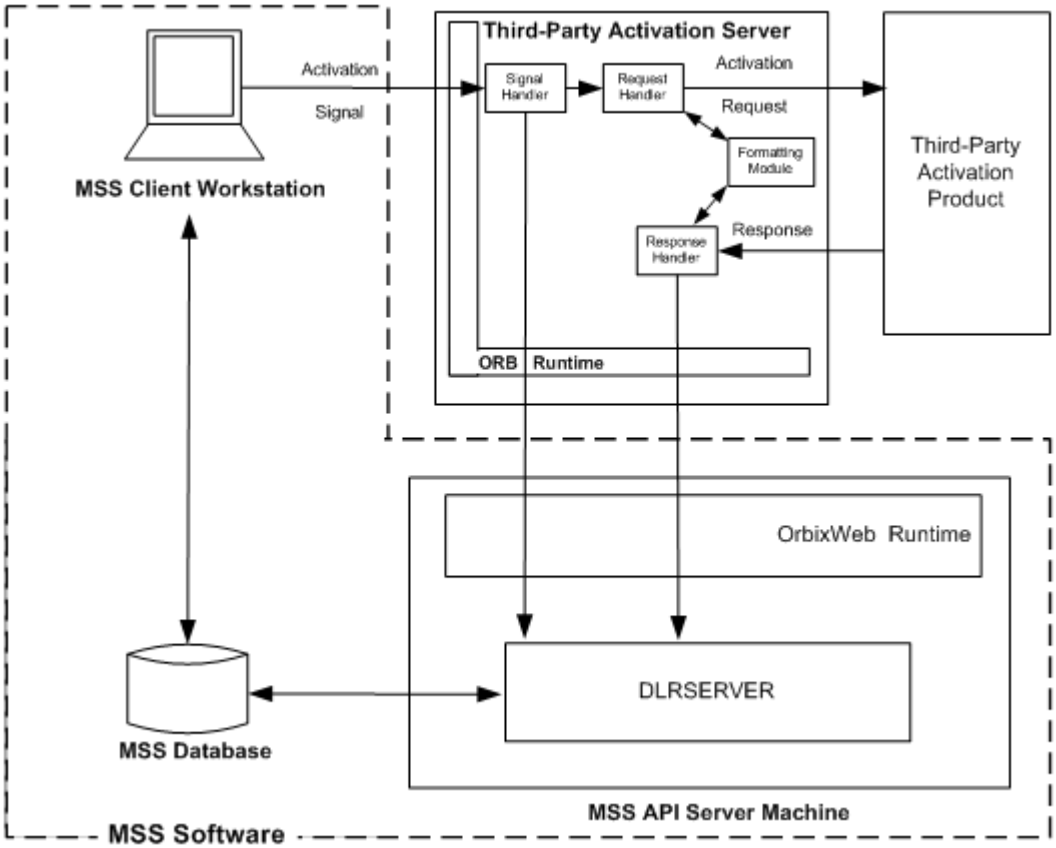
The API assumes the port calculations do not change when the physical ports are in a virtual path.

Reference Architecture

The intent of the reference architecture is to provide a logical framework to describe the various implementation concepts. It is not intended to suggest any particular application design.

Figure 12–1 shows the reference architecture for flow-through provisioning.

Figure 12–1 Reference Architecture for Flow-Through Provisioning



Gateway events are utilized to allow your activation server to integrate with the Work Management subsystem.

Signal Handler

The signal handler module implements the interfaces required to handle standard gateway events from MetaSolv Solution clients. This module is also responsible for updating gateway event status to “In Progress”.

The outbound signals sent by the MetaSolv Solution client to your activation server are the flow-through provisioning gateway events. These events are defined at the service item level. Each service item (for example, a phone line, a WATS line, or an ATM/Frame circuit) on the order will have the flow-through provisioning gateway event associated with it. As a result, when an order is processed by the Work Management subsystem, your activation server can potentially receive as many gateway events as there are service items in the order. For example, if a transport provisioning order for ASR equipment comprises six special access circuits, your activation server receives six separate gateway events from the client.

Each gateway event associated with a service item in a service request can be processed independently of the gateway events for any other service item.

Implementation should conform to the outbound signals gateway events pattern. See "Outbound Signals – Gateway Events" for more information. The signal handler module should implement a WDIDLR module with all the interfaces and operations listed in Table 2–6, " Outbound Gateway Event Operations Required For All APIs". Event status updates are performed through DLRSERVER.

Upon receiving an outbound signal conveying gateway event information from the MetaSolv Solution client, the signal handler module activates the request handler module and hands off the event information that was received. In order to avoid locking up the client, it is recommended that the signal handler should return control to the client immediately upon activating the request handler module and updating the event status.

Request Handler

The request handler module retrieves activation data from the MetaSolv Solution database by invoking the *getTransportProvisioning* operation on DLRSERVER to retrieve transport provisioning data

The operations are standard data export operations. See "Synchronous Interaction Pattern" for more information. This provisioning operation accepts a WDIEvent parameter that allows the request handler to retrieve provisioning data from the database in a single step. The request handler passes in the gateway event structure that was received from the client and DLRSERVER retrieves the required provisioning data.

It is important to understand the data types that are involved in the operation listed above. Data type definitions can be found in file WDIDLRTYPES_v5.IDL. The following Transport Provisioning data structure is returned to the caller (through callback invocation):

Example 12–1 Transport Provisioning Data Structure Example

```
typedef sequence<DLR> DLRSeq;
typedef sequence<MetaSolv::CORBA::WDIVLRTypes::VLR> VLRSeq;
struct TransportProvisioning {
    char type;   //  CIRCUIT.TYPE CHAR(1)
    DLRSeq       dlr;
    VLRSeq       vlr;
    ActivationCommandPlanSeq   activationCommandPlans ;
};
```

The ActivationCommandPlanSeq data type delivers the FTP Plan for this service item.

Formatting/Translation Module

The formatting/translation module handles two-way data translation and format conversion required for communicating with the activation product. This module's services are used by the other modules.

Response Handler

The response handler module handles responses received from the activation product. It performs the necessary reverse translation/formatting using the formatting/translation module and then determines the operation status. Based on the success or failure determination, this module updates gateway event status to "Completed" or "Errored". Design of this module depends upon factors such as the synchronous/asynchronous and online/batch nature of the interaction with activation product.

Design Considerations

To obtain the full benefit of the automated flow-through capabilities of this API, gateway events must be associated with tasks in Work Management provisioning plans. MetaSolv Solution is pre-configured with gateway templates and gateway event templates for Transport Provisioning.

See the online Help for detailed instructions on using the flow-through provisioning process.

In order to ensure that the provisioning information provided by the Transport Provisioning Activation API is sufficiently completed to be used by your network management system, care must be used when ordering the service.

The PSR module captures default values for items that have pick lists. With flow-through provisioning, defaults are also needed for editable fields. If defaults are not provided, a user would be required to manually enter the same value on every line for an order. Providing a default value in the product catalog for product specifications that are required for flow-through provisioning streamlines the ordering process.

For transport provisioning activation, the network node target identifier (TID) and the equipment port address assignment identifier (AID) in the database identify the equipment and port address. These items should either be used directly by your application or your application should maintain a cross-reference between the identifiers used by your application and the MetaSolv Solution-supplied TID and AID.

Just as the provisioning of switch features requires additional parameters, the provisioning of transport equipment requires additional parameters as well. The transport equipment for dialtone lines is usually digital loop carrier. The CLR represents the provisioning information for this type of equipment. The CLR captures the TID and the AID for the DLC equipment, which is part of the information that is required for activating the service. The TID is determined by identifying the Network Node to which the equipment belongs. The AID is determined using the assignment information that is gathered on the CLR. To provision transport equipment, additional parameters are usually required. These parameters will vary by type of equipment, by transmission rate, and by activation vendor.

13

The Trouble Management API

The Trouble Management API exposes Trouble Management subsystem functions and information that an external (third-party) application can use to:

- Create, update, and change the state of trouble tickets in the Trouble Management subsystem
- Query for trouble tickets using criteria similar to the Trouble Management subsystem's Ticket Search window
- Query to retrieve the service item identifier for a ticket to facilitate triggering of automatic testing through gateway events
- Query to retrieve various service items, customer information, and other information such as could be used to populate drop-down lists in a client application

The Trouble Management API is designed to support development of applications that integrate existing network management systems and the Trouble Management subsystem. For example, when a piece of network equipment signals an alarm, your application could use the Trouble Management API to create a trouble ticket in the Trouble Management subsystem. Periodically, or on an as-needed basis, your application could query the Trouble Management API to determine whether a trouble ticket has cleared. Until the initial trouble ticket has cleared, your application can ignore additional alarms from the faulty equipment.

The CORBA servername used by the Trouble Management API is TMSSERVER.

Functionality

Major functions for which you can use the Trouble Management API include:

- Creating new trouble tickets
- Clearing, closing, and canceling existing trouble tickets
- Creating log entries for a given trouble ticket
- Updating attributes on an existing ticket
- Querying for information about a trouble ticket
- Querying for tickets

Once a trouble ticket is created through the Trouble Management API, that ticket can be processed in the Trouble Management subsystem as if it was created using the Trouble Management subsystem. For example, Trouble Management subsystem users can refer an API-generated ticket to multiple organizations, just as if the ticket was entered using the Trouble Management subsystem.

TroubleSession Interface

Note: The Trouble Management API and the Trouble Management subsystem are separately licensed components of the Oracle Communications MetaSolv Solution product line. The Trouble Management API requires you have the Trouble Management subsystem installed. To acquire licenses to use these products, contact your Oracle representative.

WDIManager

Table 13–1 lists the operations available in the WDIManager interface of the **WDITROUBLE.IDL** file.

Table 13–1 Trouble Management API WDIManager Interface Operations

Operation	Description
startTroubleSession	Obtains the object reference for the TroubleSession
destroyTroubleSession	Terminates the TroubleSession
startTransaction	commit rollback
destroyTransaction	Terminates the Transaction
startSignal2	eventCompleted eventErrored eventInProgress eventOccurred eventTerminated
destroySignal2	Terminates the Signal2

The Trouble Management API uses the fundamental concepts of the signal handling pattern implemented by the other APIs. However, the Trouble Management API requires a different set of attribute values to uniquely identify an instance of an event within a trouble ticket. Using this variation of the signaling mechanism enables the Trouble Management API to support multiple concurrent events for a given trouble ticket.

See "Common Architecture" for more information about WDIManager.

TroubleSession Interface Operations

Table 13–2 lists the operations available in the TroubleSession interface of the **WDITROUBLE.IDL** file.

Table 13–2 Trouble Management API TroubleSession Interface Operations

Operation	WDINotification Operations
getPartyByPartyName	getPartyByPartyNameSucceeded getPartyByPartyNameFailed
getCauseCodes	getCauseCodesSucceeded getCauseCodesFailed

Table 13–2 (Cont.) Trouble Management API TroubleSession Interface Operations

Operation	WDINotification Operations
getTroubleFoundCodes	getTroubleFoundCodesSucceeded getTroubleFoundCodesFailed
getClearedCodes	getClearedCodesSucceeded getClearedCodesFailed
getServiceItemTypeCodes	getServiceItemTypeCodesSucceeded getServiceItemTypeCodesFailed
getTroubleTypeCodes2	getTroubleTypeCodes2Succeeded getTroubleTypeCodes2Failed
getInitiatingModeCodes2	getInitiatingModeCodes2Succeeded getInitiatingModeCodes2Failed
getTicketStatusCodes2	getTicketStatusCodes2Succeeded getTicketStatusCodes2Failed
getParties2	getParties2Succeeded getParties2Failed
getTicketTypeCodes2	getTicketTypeCodes2Succeeded getTicketTypeCodes2Failed
createLogEntry	createLogEntrySucceeded createLogEntryFailed
getTicketServiceItem	getTicketServiceItemSucceeded getTicketServiceItemFailed
updateTicket--Deprecated. Replaced by updateTicket_v2	updateTicketSucceeded--Deprecated. Replaced by updateTicketSucceeded_v2 operationFailed
updateTicket_v2	updateTicketSucceeded_v2 operationFailed
getTicketForUpdate-- Deprecated. Replaced by getTicketForUpdate_v2.	getTicketForUpdateSucceeded--Deprecated. Replace by getTicketForUpdateSucceeded_v2. operationFailed
getTicketForUpdate_v2	getTicketForUpdateSucceeded_v2 operationFailed
getMsgTrnkGrpServItem	getMsgTrnkGrpServItemNoDataFound getMsgTrnkGrpServItemSucceeded operationFailed
getEUSpecialTrnkGrpServItem	getEUSpecialTrnkGrpServItemNoDataFound getEUSpecialTrnkGrpServItemSucceeded operationFailed
getDSLServItem--Deprecated. This functionality is supported by the getQueryCircuits_v2 operation method in the DLR API.	getDSLServItemNoDataFound--Deprecated. getDSLServItemSucceeded--Deprecated. operationFailed

Table 13–2 (Cont.) Trouble Management API TroubleSession Interface Operations

Operation	WDINotification Operations
getInternetCircuitServItem--Deprecated. This functionality is supported by the getQueryCircuits_v2 operation in the DLR API.	getInternetCircuitServItemNoDataFound--Deprecated. getInternetCircuitServItemSucceeded--Deprecated. operationFailed
getInternetDialupServItem	getInternetDialupServItemNoDataFound getInternetDialupServItemSucceeded operationFailed
getTelephoneNumberServItem	getTelephoneNumberServItemNoDataFound getTelephoneNumberServItemSucceeded operationFailed
getCustomers	getCustomersNoDataFound getCustomersSucceeded operationFailed
getTicketForClearClose	getTicketForClearCloseSucceeded operationFailed
clearTicket	clearTicketSucceeded operationFailed
closeTicket	closeTicketSucceeded operationFailed
cancelTicket	cancelTicketSucceeded operationFailed
getTickets_v2	getTicketsNoDataFound_v2 getTicketsSucceeded_v2 operationFailed
getTicketReport_v2	getTicketReportSucceeded_v2 operationFailed
getParties_v3	getPartiesSucceeded_v3 getPartiesFailed
getCustomerAddresses	getCustomerAddressesSucceeded operationFailed
getAssignedToParties	getAssignedToPartiesSucceeded operationFailed
getEscalationMethods	getEscalationMethodsSucceeded operationFailed
createTicket_v2--Deprecated. Replaced by createTicket_v3	createTicketSucceeded_v2--Deprecated. Replaced by createTicketSucceeded_v3. operationFailed
createTicket_v3	createTicketSucceeded_v3 operationFailed

Table 13–2 (Cont.) Trouble Management API TroubleSession Interface Operations

Operation	WDINotification Operations
getNetworkElementServItem	getNetworkElementServItemNoDataFound getNetworkElementServItemSucceeded operationFailed
getNetworkSystemServItem	getNetworkSystemServItemNoDataFound getNetworkSystemServItemSucceeded operationFailed

TroubleSession Operation Descriptions

The following list contains a description of the operations available in the TroubleSession interface:

- getPartyByPartyName

  Given a Party Name and a Party Role as arguments, the getPartyByPartyName operation retrieves information for an active party. This operation may be used to get the party IDs for the Customer role (**partyRole** = "CUST'), Responsible Org (**partyRole** = 'RESP_ORG'), and Administrative Org (**partyRole** = 'ADMIN_ORG') to pass as arguments in the createTicket_v3 operation. It returns successfully only if the party and its associated party role are still active.

  If the party is an individual, the Party Name must be formatted as **Last Name, First Name**. The party name is stored in upper case in the MetaSolv Solution database, so the API converts the value passed to upper case before performing the search.

  If the operation is called to get the Party ID for a customer (**partyRole** = 'CUST'), it is possible that multiple customers may exist in the database with the same Party Name.

  Note: MetaSolv Solution does not allow multiple Responsible Organizations or Administrative Organizations to have the same name.

  If multiple party records are found for a given customer name, this operation returns an error. If this occurs, it is recommended that the customerPartyID not be passed when the createTicket_v3 operation is called. If a service item is included, the Trouble Management API automatically identifies the customer associated with the service item. The customer name can also be included in the logNotes attribute.

- getCauseCodes

  The *getCauseCodes* operation retrieves a list of cause codes defined in the MetaSolv Solution Infrastructure subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes are retrieved for drop-downs on query interfaces like the Ticket Query.

The **currentCauseCode** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentCauseCode** parameter is ignored if the **activeOnly** parameter is false.

- getTroubleFoundCodes

  The *getTroubleFoundCodes* operation retrieves a list of trouble found codes defined in the Trouble Management subsystem for a given cause code. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentTroubleFoundID** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. If passed, the **currentTroubleFoundID** must be passed as a numeric value. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentTroubleFoundID** parameter is ignored if the **activeOnly** parameter is false.

  The **causeCode** parameter limits the trouble found codes that are returned to only those that are related to this cause code. If **activeOnly** is passed as true, the cause code is required and must be a valid active or inactive cause code in the Trouble Management subsystem.

- getClearedCodes

  The *getClearedCodes* operation retrieves a list of cleared codes and could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentClearedCode** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentClearedCode** parameter is ignored if the **activeOnly** parameter is false.

- getServiceItemTypeCodes

  The *getServiceItemTypeCodes* operation retrieves a list of service item type codes supported by the Trouble Management subsystem. This operation could be used to populate a drop-down list in a user interface. The set returned will depend on the migration. See Table 13–6, " Service Item Type and Service Item Identifier" for more information.

- getTroubleTypeCodes2

  The *getTroubleTypeCodes2* operation retrieves a list of trouble type codes defined in the Trouble Management subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentTroubleTypeID** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. If passed, the **currentTroubleTypeID** must be passed as a numeric value. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentTroubleTypeID** parameter is ignored if the **activeOnly** parameter is false.

- getInitiatingModeCodes2

  The *getInitiatingModeCodes2* operation retrieves a list of initiating mode codes defined in the Trouble Management subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentInitiatingModeID** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. If passed, the **currentInitiatingModeID** must be passed as a numeric value. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentInitiatingModeID** parameter is ignored if the **activeOnly** parameter is false.

- getTicketTypeCodes2

  The *getTicketTypeCodes2* operation retrieves a list of ticket type codes defined in the Trouble Management subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentTicketTypeCode** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. Since both active and inactive codes

are returned when the **activeOnly** parameter is false, the **currentTicketTypeCode** parameter is ignored if the **activeOnly** parameter is false.

- getTicketStatusCodes2

  The *getTicketStatusCodes2* operation retrieves a list of Ticket Status Codes defined in the MetaSolv Solution Infrastructure subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the code is updated for a ticket. Both active and inactive codes should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentTicketStatusID** parameter is used to return a given inactive code along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active codes along with the ticket's current value, even if that code was inactivated after it was set on the ticket. If passed, the **currentTicketStatusID** must be passed as a numeric value. Since both active and inactive codes are returned when the **activeOnly** parameter is false, the **currentTicketStatusID** parameter is ignored if the **activeOnly** parameter is false.

  The **ticketStateCode** parameter is used to return only ticket status codes that are related to the given ticket state code. If activeOnly is passed as true, this parameter is required and must be a valid ticket state code in MetaSolv Solution. If **activeOnly** is passed as false, the ticketStateCode value is ignored. Valid ticket state codes include openActive, deferred, extreferred, cleared, closed, and canceled.

- createLogEntry

  The *createLogEntry* operation creates a log entry for a ticket. It is passed a sequence of log note strings (of no more than 2000 characters each) and creates a single log entry for the ticket. At least one log note string that does not equal spaces is required.

  Either the ticketID or documentNumber values are required as input key values. If the documentNumber is not valid, and no valid ticketID is passed, the Trouble Management API returns an exception through the *createLogEntryFailed* operation.

- getTicketServiceItem

  The *getTicketServiceItem* operation returns the ticket ID, current state code, current status ID, for a given ticket document number along with the service item type code, service item ID, and service item description if there is a service item assigned to the ticket. This query is intended to be called in response to a gateway event that is triggered when a user initiates a test of the service item on the ticket.

- updateTicket

  Deprecated. Replaced by *updateTicket_v2*.

- updateTicket_v2

  This operation updates attributes for an existing trouble ticket. This operation supports new service item types of Network Element, Network System, and Circuit/Connection. See "The updateTicket_v2 Operation" for more information.

- getTicketForUpdate

  Deprecated. Replaced by *getTicketForUpdate_v2*.

- getTicketForUpdate_v2

  This operation gets information for a ticket so that an update can be requested. It returns a structure of updateable ticket fields that may be modified and passed to the *updateTicket_v2* operation. It also returns the date and time of the report, which must be passed to the *updateTicket_v2* operation in order to verify that the ticket has not changed since the information was retrieved.

- getMsgTrnkGrpServItem

  This operation returns a list of message trunk groups. The information returned includes a circuit ID, which is the identifier passed for a message trunk group service item in the createTicket_v3 and updateTicket_v2 operations.

- getEUSpecialTrnkGrpServItem

  This operation returns a list of end-user special trunk groups. The information returned includes a two-six-code, which is the identifier passed for an end-user special trunk group service item in the createTicket_v3 and updateTicket_v2 operations.

- getDSLServItem

  Deprecated. Replaced by *getQueryCircuits_v2* in the DLR API.

- getInternetCircuitServItem

  Deprecated. Replaced by *getQueryCircuits_v2* in the DLR API.

- getInternetDialupServItem

  This operation returns a list of Internet dial-ups. The information returned includes a user ID, which is the identifier passed for an Internet dial-up service item in the createTicket_v3 and updateTicket_v2 operations.

- getTelephoneNumberServItem

  This operation returns a list of telephone numbers. The information returned includes an unformatted telephone number and a telephone number inventory id either of which can be passed as the identifier for a telephone number service item in the *createTicket_v3* and *updateTicket_v2* operations.

- getCustomers

  This operation gets a list of customers matching the criteria given by the caller. The information returned includes a party id and party address sequence which can be passed for the customer and customer address in the createTicket_v3 and updateTicket_v2 operations.

- getTicketForClearClose

  This operation retrieves clear/close information for a ticket so that a clearTicket or closeTicket operation can be requested.

- clearTicket

  This operation clears an existing trouble ticket. See "The clearTicket Operation" for more information.

- closeTicket

  This operation closes an existing trouble ticket. See "Details Concerning Use of the closeTicket Operation" for more information.

- cancelTicket

This operation cancels an existing trouble ticket. See "Details Concerning Use of the cancelTicket Operation" for more information.

- getTickets_v2

  This operation enables you to search for a trouble ticket or a collection of tickets based on a set of criteria, similar to the Ticket Search window in MetaSolv Solution. See "Details Concerning Use of the getTickets_v2 Operation" for more information.

- getTicketReport_v2

  This operation returns a ticket report. You must pass the operation either a valid document number or ticket ID. If a document number is passed, ticket ID is ignored.

- getParties_v3

  The *getParties_v3* operation retrieves a list of parties that have a given role type. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the list should contain only those codes that are currently listed in the database as active codes. Only active codes should be included in drop-downs used on interfaces where the party is updated for a ticket. Both active and inactive parties should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **currentPartyID** parameter is used to return a given inactive party along with all the active parties when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active parties along with the ticket's current value, even if that party was inactivated after it was set on the ticket. If passed, the **currentPartyID** must be passed as a numeric value. Since both active and inactive parties are returned when the **activeOnly** parameter is false, the **currentPartyID** parameter is ignored if the **activeOnly** parameter is false.

  The **partyRole** parameter is used to return only parties that have been assigned that role type. This parameter is always required.

  The enumerated type definition used for the **partyRole** parameter includes an option for CUST (Customer). However, the CUST value is not supported by the GetParties_v3 query, and results in an error if passed.

- getCustomerAddresses

  The *getCustomerAddresses* operation retrieves a list of active addresses for a given customer. This operation may be used to populate a drop-down list of addresses for a customer on a ticket. A customer address sequence number is returned with each address. The sequence number is passed along with the customer party ID to the *createTicket_v3* or *updateTicket_v2* operations to set the customer address on a ticket.

  The **customerPartyID** parameter is the party ID that identifies the customer whose addresses are to be retrieved. This is a required parameter. The **customerAddressSeq** parameter is used return a given address along with all the active addresses. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active customer addresses along with the current address set on the ticket, even if that address was inactivated after it was set on the ticket. If passed, the **customerAddressSeq** must be passed as a numeric value.

- getAssignedToParties

The *getAssignedToParties* operation retrieves a list of active employees that are associated with either a responsible organization or an administrative organization. This operation may be used to populate a drop-down list of Responsible Organization Assigned To parties or Administrative Organization Assigned To parties on a ticket.

The **orgPartyID** parameter is the party ID that identifies the responsible organization or administrative organization whose employees are to be retrieved. This is a required parameter. The **assignedToPartyID** parameter is used return a specific Assigned To party along with all the active employees. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active employees along with the current Assigned To party on the ticket, even if that party was inactivated after it was set on the ticket.

- getEscalationMethods

  The *getEscalationMethods* operation retrieves a list of escalation methods defined in the Trouble Management subsystem. This operation could be used to populate a drop-down list in a user interface.

  You can use the **activeOnly** Boolean parameter to specify whether or not the operation should retrieve only active escalation methods. Only active values should be included in drop-downs used on interfaces where the field is updated for a ticket. Both active and inactive values should be retrieved for drop-downs on query interfaces like the Ticket Query.

  The **escalationMethodID** parameter is used to return a given inactive escalation method along with all the active values when the **activeOnly** parameter is true. This parameter is optional. It enables you to populate a dropdown field on a ticket that includes all the active escalation methods along with the ticket's current value, even if that value was inactivated after it was set on the ticket. If passed, the **escalationMethodID** must be passed as a numeric value. Since both active and inactive escalation methods are returned when the **activeOnly** parameter is false, the **escalationMethodID** parameter is ignored if the **activeOnly** parameter is false.

- createTicket_v2

  Deprecated. Replaced by *createTicket_v3*.

- createTicket_v3

  This operation creates a ticket. The createTicket_v3 operation supports the service item types of Network Element, Network System, and Circuit/Connection. See "The createTicket_v3 Operation" for more information.

- getNetworkElementServItem

  This operation returns a list of network elements. The information returned includes a a service item ID and name, which are the identifiers passed for a network element service item in the createTicket_v3 and updateTicket_v2 operations.

- getNetworkSystemServItem

  This operation returns a list of network systems. The information includes a service item ID and name, which are the identifiers passed for a network system service item in the createTicket_v3 and updateTicket_v2 operations.

Trouble Management API IDL Files

The IDL files that describe the operations and data structures that comprise the Trouble Management API are:

- WDITROUBLE.IDL
- WDITROUBLETYPES.IDL
- WDITROUBLETYPES2.IDL
- WDITROUBLETYPES_v3.IDL
- WDITROUBLETYPES_v4.IDL
- WDI.IDL
- WDIUTIL.IDL

Process Flows

This section contains a sample process flow for a solicited message. Use the sample flow as a template for developing your own process flows.

Refer to the next section for the process flow used when the Trouble Management API is the client.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. In this case, the Trouble Management API plays the role of the client, and your application plays the role of the server.

Table 13–3 lists the interfaces and operations that your application implements using the IDL file provided with the Trouble API.

Table 13–3 Trouble Management API Solicited Message Operations

Interface	For Implementing These Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	N/A
WDISignal	eventOccurred eventTerminated
WDIInSignal	N/A

Sample: Solicited Message Process Flow

When the Trouble Management API is the client, the overall process flows as follows:

1. The Trouble Management API requests a WDIRoot object reference from your application. The request is routed through the ORB.
2. Your application instantiates a WDIRoot and returns a WDIRoot object
3. The Trouble Management API invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.

4. The Trouble Management API invokes the *startSignal2* operation of the WDIManager interface to get a WDISignal2 object reference.
5. The Trouble Management API invokes the *eventOccurred* operation of the WDISignal2 interface, passing a WDIEvent2 structure to notify your application that an event registered to them has occurred within MetaSolv Solution.
6. The Trouble Management API invokes the *destroySignal2* operation of the WDIManager interface.
7. The Trouble Management API invokes the *disconnect* operation of the WDIRoot interface.
8. Once your application completes processing, possibly involving additional unsolicited messages to the APIs, your application connects to the MetaSolv Solution Application Server and follows the same process described above for the API client with the exception that the *eventCompleted/Errored* operations are invoked passing the original WDIEvent2 structure.

If your application encounters an error, it throws a WDIExcp as defined by the IDL. The Trouble Management API handles CORBA system exceptions and WDIExcp exceptions.

Unsolicited Messages

An unsolicited message is a message initiated by your application. In this case, the Trouble Management API plays the role of the server and your application plays the role of the client with the exception of the callback processing.

Table 13–4 lists the interfaces and operations that the Trouble Management API implements.

Table 13–4 Trouble Management API Unsolicited Message Operations

Interface	Implemented Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	commit rollback

Sample Flows for Business Tasks

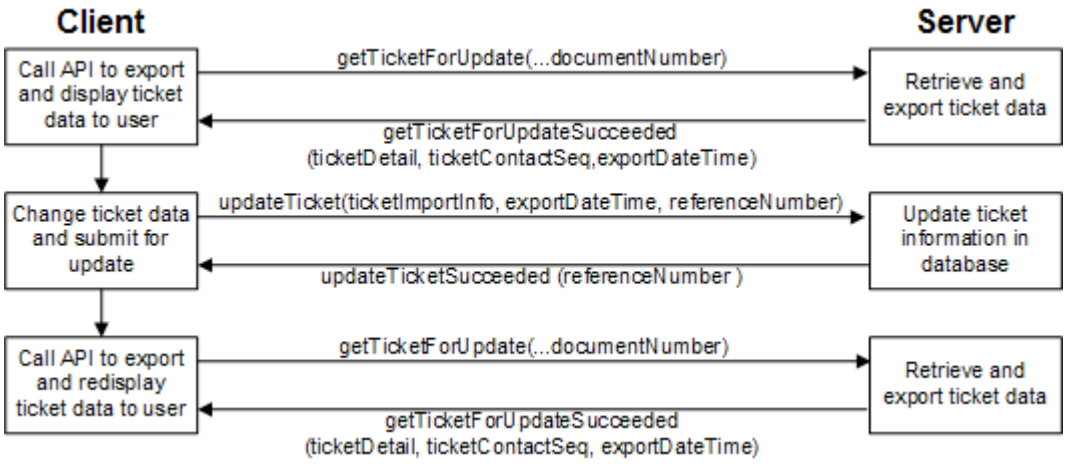
This section provides a few sample flows for business tasks.

Process Flow for Updating a Trouble Ticket

This process flow demonstrates how your client application and the Trouble Management API server must interact to update a trouble ticket.

Figure 13–1 illustrates the process flow for updating a trouble ticket.

Figure 13–1 Process Flow for Updating a Trouble Ticket



The process flow for updating a trouble ticket is as follows:

1. The client calls the getTicketForUpdate operation to retrieve trouble ticket information.
2. The server calls the getTicketForUpdateSucceeded operation to return a TicketInfoForUpdate structure, which includes one structure of attributes that are updateable and another structure of attributes that are read-only. The structure also includes the date and time of the export.
3. The client makes modifications to the updateable data and calls the updateTicket operation, passing it the modified UpdateableTicketInfo structure. In addition to the WDITransaction and WDINotification objects, the updateTicket operation is passed the following parameters:
   - The TicketImportInfo structure. This structure includes the ticket's document number and ticket ID.

     Note: You must pass either document number or ticket ID. If document number is passed, ticket ID is ignored. If you pass neither, an exception is returned.

   - The UpdateableTicketInfo structure with any modifications.
   - A ServiceItemSeq sequence, which is populated only if the service item is changed. Only one ServiceItem structure can be passed. If more than one structure is passed, an exception is returned.
   - A LogNoteInfoSeq sequence, which is included if the update includes log notes.
   - A duplicateTicketAllowed Boolean that indicates whether a change to the service item is allowed if another open ticket is found for the same service item. Both the ticket found and the ticket being updated have a ticket type that identifies repeat and chronic trouble (as defined for the ticket type in the MetaSolv Solution Infrastructure). If it is set to false, an exception is returned if another open ticket exists. The client can initially set it to false, and if an exception is returned, present a message to the user asking if they wish to create the ticket anyway (similar to the functionality in MetaSolv Solution). If

so, the UpdateTicket operation can be called again with **duplicateTicketAllowed** set to true.

- The export date and time that was returned from the export. The Trouble Management API server uses this date and time to throw an exception if the ticket was updated by some other process since the ticket information was first exported.

4. The Trouble Management API server processes the update and indicates success or failure by calling either the updateTicketSucceeded operation or the operationFailed operation on WDINotification object.
5. Upon the successful update of the ticket, the client application should refresh its user interface by again retrieving the ticket attributes using the getTicketForUpdate operation. This action resets the export date and time.

Process Flow for Clearing a Trouble Ticket

This process flow demonstrates how your client application and the Trouble Management API server must interact to clear a trouble ticket. Clearing a ticket is done when the trouble has been resolved, and you are waiting for the customer to verify that the service has been restored so the ticket may be closed.

1. The client calls the getTicketForClearClose operation to retrieve trouble ticket information.
2. The server calls the getTicketForClearCloseSucceeded operation to return a clearCloseTicketExportInfo structure, which includes a structure of attributes that are updateable, the ticket's document number, and the ticket's unique Trouble Management subsystem ID.
3. The client populates the ClearCloseTicketImportInfo structure, then calls the clearTicket operation. In addition to the WDITransaction and WDINotification objects, the clearTicket operation is passed the following parameters:
   - The ClearCloseTicketImportInfo structure. This structure includes the ticket's document number and ticket ID.

     Note: You must pass either document number or ticket ID. If document number is passed, ticket ID is ignored. If you pass neither, an exception is returned.

   - The export date and time that was returned from the export. The Trouble Management API server uses this date and time to throw an exception if the ticket was updated by some other process since the ticket information was first exported.
   - An arbitrary reference number supplied by your client application that identifies the transaction. If the clearTicket operation is successful, the Trouble Management API returns this reference number through the clearTicketSucceeded notification.
4. The Trouble Management API server processes the operation and indicates success or failure by calling either the clearTicketSucceeded operation or the clearTicketFailed operation on WDINotification object.

Process Flow for Closing a Trouble Ticket

This process flow demonstrates how your client application and the Trouble Management API server must interact to close a trouble ticket. Closing a ticket occurs when the trouble is resolved and the customer verifies the service is restored.

1. The client calls the getTicketForClearClose operation to retrieve trouble ticket information.
2. The server calls the getTicketForClearCloseSucceeded operation to return a clearCloseTicketExportInfo structure, which includes a structure of updateable attributes, the ticket's document number, and the ticket's unique Trouble Management subsystem ID.
3. If the ticket is a parent (that is, **ParentChildInd** = Y), the client should prompt the user to determine whether the child tickets should be closed with the parent. If so, the **processChildTickets** field on the ClearCloseTicketImportInfo structure should be passed as TRUE.
4. The client populates the ClearCloseTicketImportInfo structure then calls the closeTicket operation. In addition to the WDITransaction and WDINotification objects, the closeTicket operation is passed the following parameters:
   - The ClearCloseTicketImportInfo structure. This structure includes the ticket's document number and ticket ID.

     Note: You must pass either document number or ticket ID. If document number is passed, ticket ID is ignored. If you pass neither, an exception is returned.

   - The export date and time that was returned from the export. The Trouble Management API server uses this date and time to throw an exception if the ticket has been updated by some other process since the ticket information was exported.
   - An arbitrary reference number supplied by your client application that identifies the transaction. If the closeTicket operation is successful, the Trouble Management API returns this reference number through the closeTicketSucceeded notification.
5. The Trouble Management API server processes the operation and indicates success or failure by calling either the closeTicketSucceeded operation or the closeTicketFailed operation on WDINotification object.

Process Flow for Canceling a Trouble Ticket

1. This process flow demonstrates how your client application and the Trouble Management API server must interact to cancel a trouble ticket. The cancelTicket operation changes the ticket's state to Canceled.
2. Your client application populates the CancelTicketImportInfo structure then calls the cancelTicket operation.

   Note: You must pass either document number or ticket ID. If document number is passed, ticket ID is ignored. If you pass neither, an exception is returned.

3. The server processes the cancellation of the ticket. If the cancellation fails, the server calls the operationFailed operation on the WDINotification object. If the operation is successful, the server calls the cancelTicketSucceeded operation. In this case the server also creates a log entry with an audit note for each attribute changed when canceling the ticket, similar to the notes generated by the Trouble Management subsystem.

> **Note:** If the cancellation is successful, and the ticket is a parent ticket (that is, the ticket's **ParentChildInd** = Y), then the associated child tickets are also automatically canceled along with the parent ticket. Because of the potential impact of inadvertently canceling many child tickets, you may wish to have your client application display a warning that all child tickets will also be canceled and request that your user confirm the action.

Using the Service Item Test Button Functionality

The service item test button functionality of the Trouble Management subsystem requires that your external application follow a particular sequence of events once it receives a signal that indicates that the **Test SI** button has been clicked within the Trouble Management subsystem. In order to use the service item test button functionality:

1. Your external application must receive the gateway event signal that indicates that the **Test SI** button has been clicked. The gateway event signal contains the trouble ticket's document number.
2. Your application must use the getTicketServiceItem operation to retrieve the service item ID on the trouble ticket.

> **Note:** Your application may be able to make use of other API operations that can return the TID, AID, or other identifier that uniquely identifies the service item.

3. Once the test is completed, your application should update the status of the gateway event to Completed. You can also have your application use the createLogEntry operation to write a trouble ticket log entry that describes the results of the test.

Implementation Concepts

See the following for more information on the Trouble Management API:

- Developing Applications Using the APIs
- HelloAPI: Sample Application that Exports Data

Interaction Life Cycle

1. The external application sends a message to the Trouble Management API through the CORBA implementation. The message consists of the operation requested, the data required by the operation's parameters, and a WDINotification object.
2. The Trouble Management API executes the requested operation.

3. Based on the result of the operation, the Trouble Management API determines the appropriate response to return to the external application then:
   - If the operation was successful, the Trouble Management API invokes the corresponding **succeeded** operation on the WDINotification object. The parameters passed with the invocation include any data that is appropriate for the response.
   - If the operation was successful but the database contains no records that match the criteria you specified, and the notification operations for that query include a NoData operation, the Trouble Management API invokes the NoData operation on the WDINotification object.
   - If the operation was unsuccessful, the Trouble Management API invokes the corresponding **failed** operation on the WDINotification object. The parameters that the Trouble Management API passes when invoking the operation include appropriate error messages.

Session User ID Can Be Used to Verify Workforce Employee

When your client application calls the connect operation on the WDIManager object, the ConnectReq structure must contain a valid User ID or the connect operation fails. For the Trouble Management API, that User ID must be the workforce User ID for a valid workforce employee. Workforce employees are set up through the MetaSolv Solution Workforce Employee window. This is a different process than setting up an employee with a MetaSolv Solution User ID, and an employee's MetaSolv Solution User ID can be different from their workforce User ID.

> **Note:** The WDIManager object is a common object used by all the MetaSolv Solution APIs, and the ConnectReq structure was designed to support MetaSolv Solution User IDs and passwords. Unlike MetaSolv Solution User IDs, the Trouble Management subsystem's workforce User IDs do not have a password. Therefore, when populating the ConnectReq structure for use with the Trouble Management API, you can populate the **Password** field with an empty string,

For certain Trouble Management operations where an audit trail is desirable, the Trouble Management API uses this session User ID instead of the global User ID that is specified in the MetaSolv Solution Application Server's **gateway.ini** file. The session User ID identifies the workforce user who made the changes to the ticket.

The Trouble Management API operations that require the session User ID are:

- cancelTicket
- clearTicket
- closeTicket
- createLogEntry
- createTicket_v3
- updateTicket_v2

When your client application successfully calls one of these TroubleSession interface operations, the API stores the session User ID (workforce User ID) in the audit notes. The User ID identifies the user who made the changes to the ticket. This mechanism

permits you to build client applications that implicitly verify that the requesting user is authorized to perform the critical trouble management actions shown in the list above. If you prefer not to use this verification approach, you should pass a session User ID that is known to be set up in the Trouble Management subsystem as a workforce employee.

Date Field Types

Date fields of type UTCDate are in Coordinated Universal Time (UTC) which can be considered equivalent to GMT. Date fields of type MSVDate are in database server time.

The createTicket_v3 Operation

This section provides information about the createTicket_v3 operation.

Import Ticket Attributes

The createTicket_v3 and updateTicket_v3 operations share the structure UpdateableTicketInfo. Upon successful completion of the createTicket_v3 operation, the ticket ID and the document number are returned with the notification createTicketSucceeded_v3.

The createTicket_v3 operation also accepts an unlimited sequence of log notes. Each note can be up to 2,000 characters long. The Trouble Management subsystem displays these log notes as API Additional Info log notes. This replaces the AdditionalTroubleInfo sequence in the previous release.

The createTicket_v3 operation supports the service item types of Network Element, Network System, and Circuit/Connection.

Required Fields in createTicket_v3 Request

The following fields within the UpdateableTicketInfo structure must be populated when any create ticket is requested:

- Trouble Detection Date (**troubleDetectionDate**)
- Ticket Type Code (**ticketTypeCode**)
- Initiating Mode ID (**initiatingModeID**)
- Ticket Status ID (**ticketStatusID**)
- Priority Level ID (**priorityLevelID**)
- Responsible Org Party ID (**responsibleOrgPartyID**)
- Administrative Org Party ID (**administrativeOrgPartyID**)
- Intrusive Testing Authorized Indicator (**intrusiveTestingAuthInd**)
- Billing Type Code (**billTypeCd**)

Business Rules in Processing createticket_v3 Request

The following items list business rules used in validating and processing the createTicket_v3 request:

- All code and ID fields must exist and be active.
- The Trouble Detection Date must be on or before the ticket open date.
- The Ticket Status ID must be valid for the “openActive” ticket state.

- Priority Level values are 0, 1, 2, and 3.
- Severity Level values are 0, 1, 2, and 3.
- Reported By and Ticket Contact Access Numbers are now taken as a string, instead of in the previous release's TelephoneNumber structure.
- Reported By and Ticket Contact Access Numbers can only contain numeric characters if the MetaSolv Solution Enable NPA/NXX Contact Telephone Number Formatting preference is "Yes." If this preference is "Yes" and the access number is not numeric, the access number is not stored with the contact. Instead, a log note is added to provide the telephone number information.
- The Reported By and Ticket Contact Access Numbers are only stored when a contact name is given. If the contact name is not given, a log note indicates that the contact access number could not be stored. The log not includes the imported access number.
- The Customer Address Sequence can only be specified when the Customer Party ID is specified.
- Responsible Org Assigned To Accepted Indicator must be populated with Y or N if the Responsible Org Assigned To Party ID is populated.
- Administrative Org Assigned To Accepted Indicator must be populated with Y or N if the Administrative Org Assigned To Party ID
- Office Network Location must be a valid location.
- Billing Type Code valid values are "bill" and "nonBill."
- The Next Customer Status Date cannot be prior to the current date (time is not considered).
- The Service Item Sequence (ServiceItemSeq) within the TicketImportInfo structure can either contain 0 or 1 instances of the ServiceItem structure.
- The Log Note Information Sequence (LogNoteInfoSeq) can contain any number of entries. The log note text can only contain a maximum of 2000 character each.
- These fields, if populated, are required to be numeric:
  - Responsible Org ID (**responsibleOrgPartyID**)
  - Responsible Org Assigned To ID (**respOrgAssignedToPartyID**)
  - Administrative Org Party ID (**administrativeOrgPartyID**)
  - Administrative Org Assigned To Party ID (**adminOrgAssignedToPartyID**)
  - Customer Party ID (**customerPartyID**)
  - Escalation Method ID (**escalationMethodID**)
  - Initiating Mode ID (**initiatingModeID**)
  - Ticket Status ID (**ticketStatusID**)
  - Trouble Found ID (**troubleFoundID**)
  - Trouble Type ID (**troubleTypeID**)
  - Customer Status Minutes (**customerStatusMinutes**)
  - ETTR (**ettrSeconds**)
  - Priority Level (**priorityLevelID**)
  - SeverityLevel (**severityLevelID**)

See "Trouble Management Operational Differences" for more information.

Notifications Upon Ticket Creation

When the createTicket_v3 operation is used to change the **Responsible Org**, **Administrative Org**, **Resp Org Assigned To**, or **Admin Org Assigned To** change, all appropriate notifications are generated, just as if the change had been made from within the MetaSolv Solution Trouble Management subsystem.

Escalation Levels for createTicket_v3 Request

The createTicket_v3 operation does not support input for escalation levels for the **Responsible Org** and **Administrative Org**. It also does not support the input of other escalation organizations on a ticket.

If the input **Responsible Org** and/or **Administrative Org** have an escalation profile defined for the input **Escalation Method** (defined in the organization's escalation profile in Infrastructure), the initial escalation level for the organization is defaulted on the new ticket by the API. If the input **Escalation Method** has a default escalation organization defined in Infrastructure, that escalation organization and its initial escalation level is defaulted on the new ticket by the API.

Ticket Linkage

Creating a parent-child relationship with another ticket through the Trouble Management API is not supported.

Creating Duplicate Tickets

The **duplicateTicketAllowed** Boolean field in the TicketImportInfo structure determines whether the API allows setting the service item on a ticket if an open ticket already exists on the that service item, and both tickets have a ticket type that identifies repeat and chronic trouble.

Customer Must Be Passed as a Party ID

In the Trouble Management subsystem, users optionally enter the customer name directly instead of selecting the customer from the Customer Search window. If the customer is not found, the Trouble Management subsystem displays an error when the ticket is saved. The createTicket_v3 operation requires that the customer be passed in the form of a party ID if a customer is being specified. A client may still allow the user to enter the customer name directly and determine the ID by calling the getPartyByPartyName operation. That operation returns a party ID which can then be passed to the createTicket_v3 operation.

Customer is Defaulted Based On the Service Item

If the service item is changed, and there is no customer on the ticket, the API defaults the customer to the customer associated in the MetaSolv Solution database with the service item. If the defaulted customer has only one billing address, the address is also defaulted.

Non-inventoried Service Items Are Not Created

A non-inventoried service item is not created if the service item on a ticket cannot be found in inventory. If the service item cannot be found, the new service item type is set on the ticket, but the service item description is set to null. A log note is created stating that the service item could not be found. The log note includes the service item identifier information passed in the ServiceItem structure.

Certain Codes Are Passed as ID Values

Changes to the following codes on a ticket are passed in the form of their numeric ID values, not the code directly. Trouble Management API queries that return the numeric ID and the code are available for each. This enables you to populate dropdown fields on the client application.

- Escalation Method ID
- Initiating Mode ID
- Ticket Status ID
- Trouble Found ID
- Trouble Type ID

Ticket Dates and Times Are Imported in GMT

The Trouble Management API assumes that all dates imported through the createTicket_v3 operation are in GMT. It is the responsibility of the client application to convert any imported dates from local time to GMT.

Telcordia Preference and Trouble Management API

MetaSolv Solution uses its Telcordia preference to determine if circuit identifier fields should be formatted according to Telcordia specifications (for example, having the proper number of spaces between virgules). The setting of the Telcordia preference has no effect when you use the Trouble Management API to specify a circuit for the createTicket_v2, createTicket_v3, updateTicket, or updateTicket_v2 operations, and the service item type is one of the types shown in Table 13–5. In such a case, the Trouble Management API searches the database for the value in the corresponding field as a formatted circuit. If the API does not find that value as a formatted circuit, the API searches again for that value as an unformatted circuit using the input provided in the operation's parameters.

Table 13–5 lists the service item types and the field names.

Table 13–5 Field Formatting

Service item type	Field name
Circuit/Connection	CircuitConnectionID
Internet Circuit	InternetCircuitIdent
Internet DSL	DSLCircuitIdent
Message Trunk Group	MsgTrunkGroupIdent

Setting or Changing the Affected Service Item On a Trouble Ticket

The service item on a ticket may be set or changed through the createTicket_v3 or updateTicket_v2 operations by passing a single ServiceItem structure in the ServiceItemSeq sequence. If no change is to be made to the service item, no structure should be sent. Only one structure may be passed in the sequence.

If the API finds the service item in the MetaSolv Solution inventory, the service item type and appropriate service item description are set on the ticket. If the service item cannot be found, the API processes the ticket creation or update without an error, but sets only the service item type on the ticket and writes a log note indicating that the service item could not be found. The log note includes the service item information

passed for the service item type. The API does not create non-inventoried service items.

Passing the Service Item Type and Service Item Identifier

The ServiceItem structure includes a service item type attribute and a set of service item identifier attributes. The service item type is an enumerated attribute that categorizes the service items supported by the Trouble Management System. The API uses the service item type to determine which service item identifier to use in attempting to find the service item. Only the appropriate service item identifier is used, and all other information passed is ignored. Table 13–6 lists the service item type values and their corresponding service item identifiers from the ServiceItem structure. One exception to this is the serviceItemID field. It can be used to specify any one of the following service item types.

Table 13–6 Service Item Type and Service Item Identifier

Service item type	Enumerated value	Service item identifier
Equipment	EQUIPMENT	See "Identifying an Equipment Service Item Type" for more information.
Circuit/Connection	CIRCUIT	See "Identifying a Circuit/Connection Service Item Type" for more information.
Message Trunk Group	MSG_TRNKGRP	**msgTrunkGroupIdent** - This is the circuit ID of the message trunk group.
End User Special Trunk Group	EUS_TRNKGRP	**eusTrunkGroupIdent** - This is the two six code of the end user special trunk group
Telephone Number	TELNBR	See "Identifying a Telephone Number Service Item Type" for more information.
Internet Dial-Up	INTRNTDLP	**InternetDialupIdent** - This is the user ID of the internet dial up service
Internet Circuit	INTRNTCKT	**InternetCircuitIdent** - This is the circuit ID of the internet circuit. After migration to the new MSS graphical format, this service item type moves to the Circuit/Connection service item type.
Internet DSL	BWCKT	**DSLCircuitIdent** - This is the circuit ID of the Digital Subscriber Line bandwidth circuit. After migration to the new MSS graphical format, this service item type moves to the Circuit/Connection service item type.

Identifying a Circuit/Connection Service Item Type

When a Trouble Management subsystem user creates a trouble ticket on a service item that has a service item type of CIRCUIT, it enables you to identify the faulty circuit by using the circuit ID. The Trouble Management API also enables you to use the circuit ID to identify the faulty circuit.You can use the **serviceItemId** field to specify a circuit or a connection, in addition to the fields in the CircuitConnectionInfo structure, which was previously called CircuitInfo.The circuit ID can be retrieved from the getQueryCircuits_v2 operation in the DLR API.

In addition, the Trouble Management API enables you to identify the faulty circuit by using port information that is associated with the circuit's port address. The port address information includes:

- **Target Identifier (TID):** The TID identifies a group of equipment associated as part of a system or network element. In MetaSolv Solution, the TID information is maintained on the Node tab of the Network Element Properties window.
- **Access Identifier (AID):** The AID identifies the port address on a piece of equipment within the network element identified by the TID. In MetaSolv Solution, the AID information is stored as the concatenated node address for the port address to which the circuit is assigned.

Using port address information enables you to create a trouble ticket on a circuit when an alarm is triggered on a port address monitored by a fault management product.

> **Note:** Using port address information for Circuit/Connection service items enables you to use the Outage report to identify all customers affected by the outage and contact the customers proactively to advise them of the trouble. You can generate the Outage report from the Active Ticket Queue window in the Trouble Management subsystem.

Identifying an Equipment Service Item Type

When you use the Trouble Management API to create a trouble ticket on a service item that has a service item type of EQUIPMENT, the Trouble Management API enables you to use one of four methods to identify the faulty equipment:

- Equipment ID: The equipment ID for an installed piece of equipment. The equipment ID is retrieved from the queryEquipInstall_v2 operation in WDIEquipment.
- Equipment Name: The equipment name for an installed piece of equipment. The equipment name is maintained in the Name field on the Equipment tab of the Equipment window.
- Serial Number: The serial number for an installed piece of equipment. The serial number is maintained in the Serial Number field on the Equipment tab of the Equipment window.
- Serial Number and COMMON LANGUAGE Equipment Identifier (CLEI) Code: If serial numbers are not unique among the vendors of your installed equipment, you can pass the serial number and CLEI code for the faulty equipment. Uniqueness of CLEI codes is enforced by Telcordia Technologies (formerly Bellcore). However, the CLEI code alone does not sufficiently identify a single piece of equipment. The CLEI code is maintained in the **CLEI** code field on the Equipment Spec tab of the Equipment Spec window.

> **WARNING: The Trouble Management API can use these methods to identify a specific piece of equipment only if you maintain a unique equipment name or unique serial number values for each installed piece of equipment. The Name and Serial Number fields on the Equipment Maintenance window and the CLEI code field on the Equipment Spec tab of the Equipment Spec window are not required fields. Also, MetaSolv Solution does not enforce any validation on the Name and Serial Number fields to ensure that they are unique.**

Neither the Trouble Management subsystem nor the Trouble Management API support creating a trouble ticket on equipment that has a service item type of

EQUIPMENT at the port address level. The lowest level at which you can create a trouble ticket for Equipment service items is the card on which the port address resides. If you need to create trouble tickets for Equipment service items at a lower level than the card, a work-around method is to create the ticket with a service item type of CIRCUIT instead of EQUIPMENT and pass the TID and AID associated with the port address.

Identifying an Network Element Service Item Type

You can specify a Network Element service item type by specifying the service item type as Element in the **servItemType** field. You can then specify the specific element by either populating the **networkElementName** field with the network element name, or you can use the **serviceItemId** field. The **serviceItemId** field is preferred, because the **networkElementName** field can refer to more than one element. Both fields are returned by the getNetworkElementServItem query operation.

Identifying a Network System Service Item Type

You can specify a Network System service item type by specifying the service item type as System in the **servItemType** field. You can then specify the specific system by either populating the **networkSystemShortName** field with the unique network system short name, or you can use the **serviceItemId** field. Both fields are returned by the getNetworkSystemServItem query operation.

Identifying a Telephone Number Service Item Type

When you use the Trouble Management API to create a trouble ticket on a service item that has a service item type of Telephone Number, the Trouble Management API enables you to use one of two methods to identify the appropriate number:

- **UnformattedTelephoneNumber:** The telephone number in a single string format, without containing any formatting characters (that is, it should be all numeric characters) for a telephone number.
- **TelephoneNbrInvId:** The number inventory ID for telephone number. The number inventory ID is retrieved from the operation within the Trouble API.

Clearing the service item from a ticket

A service item may be cleared from an existing ticket by passing a ServiceItem structure with the service item type set to “none”. No service item identifiers need to be populated in that case. The API clears both the service item type and the service item description from the ticket.

The updateTicket_v2 Operation

This section provides information about the updateTicket_v2 operation.

Updateable Ticket Attributes

When executed successfully, the getTicketForUpdate_v2 operation returns a TicketInfoForUpdate structure which contains an UpdateableTicketInfo structure and a ReadOnlyTicketInfo structure. The trouble ticket attributes you can change through the updateTicket_v2 operation are contained in the UpdateableTicketInfo structure. For each ticket attribute that is changed, a log entry is created with an audit note.

The updateTicket_v2 operation also accepts an unlimited sequence of log notes. Each note can be up to 2,000 characters long. The Trouble Management subsystem displays

log notes along with any audit notes that are generated by the Trouble Management API.

> **Note:** In the Trouble Management subsystem, when the **Responsible Org**, **Resp Org Assigned To**, **Administrative Org**, or **Admin Org Assigned To** are changed on a ticket, a log note is required. The Trouble Management API does not require a log note when these fields are changed through the *updateTicket_v2* operation. If necessary, this may be enforced by the client.

ExportDateTime Field is Used to Check Concurrency

The export date and time (aExportDateTime) returned by the getTicketForUpdate_v2 succeeded notification (getTicketForUpdateSucceeded)is in the database server's time zone. You passed this information back unchanged in the updateTicket _v2 operation and the API uses it to verify that the ticket has not been updated since the read operation in the getTicketForUpdate_v2. If the ticket has been updated after the getTicketForUpdate_v2 read, then an exception is returned.

Required Fields in updateTicket Request

The following fields within the updateableTicketInfo structure must be populated when any update is requested:

- Trouble Detection Date (troubleDetectionDate)
- Ticket Type Code (ticketTypeCode)
- Initiating Mode Id (initiatingModeID)
- Ticket Status Id (ticketStatusID)
- Priority Level Id (priorityLevelID)
- Responsible Org Party Id (responsibleOrgPartyID)
- Administrative Org Party Id (administrativeOrgPartyID)
- Intrusive Testing Authorized Indicator (intrusiveTestingAuthInd)
- Billing Type Code (billTypeCd)

Business Rules in Processing updateTicket_v2 Request

The following items list business rules used in validating and processing the updateTicket_v2 request:

- Closed tickets may not be edited.
- All code and id fields must exist and be active.
- The Trouble Detection Date must be on or before the ticket open date.
- The Ticket Status Id must be valid for the current ticket state.
- Priority Level values are 0, 1, 2, and 3.
- Severity Level values are 0, 1, 2, and 3.
- Contact Access Numbers can only contain numeric characters if the "Enable NPA/NXX Contact Telephone Number Formatting" preference is Yes. Also, this can only be stored when a contact name is given.

- The Customer Address Sequence can only be specified when the Customer Party Id is specified.
- Responsible Org Assigned To Accepted Indicator must be populated with Y or N if the Responsible Org Assigned To Party Id is populated.
- Administrative Org Assigned To Accepted Indicator must be populated with Y or N if the Administrative Org Assigned To Party Id
- Office Network Location must be a valid location.
- Billing Type Code valid values are “bill” and “nonBill.”
- Cause Code field is required if the ticket is in a “cleared” state.
- Trouble Found Id field is required if the ticket is in a “cleared” state.
- If Trouble Found Id field is populated then the Cause Code must be populated.
- The Trouble Found id field must be associated to the Cause Code.
- Cleared Code is required if the ticket is in a “cleared” state.
- The Defer Until Date may be changed only if the ticket is in a Deferred state, and this date cannot be prior to the current date (time is not considered).
- The Next Customer Status Date cannot be prior to the current date (time is not considered).
- The Service Item Sequence (ServiceItemSeq) within the TicketImportInfo structure can either contain 0 or 1 instances of the ServiceItem structure. If the Service Item Sequence is not given, then it is assumed that it has not changed.
- The Log Note Information Sequence (LogNoteInfoSeq) can contain any number of entries. The log note text can only contain a maximum of 2000 character each.
- These fields, if populated, are required to be numeric:
  - Responsible Org Id (**responsibleOrgPartyID**)
  - Responsible Org Assigned To Id (**respOrgAssignedToPartyID**)
  - Administrative Org Party Id (**administrativeOrgPartyID**)
  - Administrative Org Assigned To Party Id (**adminOrgAssignedToPartyID**)
  - Customer Party Id (**customerPartyID**)
  - Escalation Method Id (**escalationMethodID**)
  - Initiating Mode Id (**initiatingModeID**)
  - Ticket Status Id (**ticketStatusID**)
  - Trouble Found Id (**troubleFoundID**)
  - Trouble Type Id (**troubleTypeID**)
  - Customer Status Minutes (**customerStatusMinutes**)
  - ETTR (**ettrSeconds**)
  - Priority Level (**priorityLevelID**)
  - SeverityLevel (**severityLevelID**)

See "Trouble Management Operational Differences" for more information.

Notifications Upon Ticket Update

When the updateTicket_v2 operation is used to change the **Responsible Org**, **Administrative Org**, **Resp Org Assigned To**, or **Admin Org Assigned To** change, all appropriate notifications are generated, just as if the change had been made from within the MetaSolv Solution Trouble Management subsystem.

Ticket Linkage and Ticket Update

If the updated ticket is linked in a common cause relationship as a parent ticket, the updateTicket_v2 operation synchronizes the child ticket(s) with the parent ticket. The Trouble Management API does not include functionality to link or unlink tickets. It only keeps the parent and child tickets synchronized when the parent ticket changes.

Attributes on a child ticket cannot be explicitly altered by an updateTicket_v2 request on the child ticket itself. These updates must be made to the parent ticket. These child ticket attributes are automatically updated when the corresponding attribute changes on the parent ticket:

- Ticket Status
- Responsible Organization
- Administrative Organization
- Office Network Location
- Priority Level
- Severity Level
- ETTR
- Trouble Description
- Trouble Detection Date
- Admin Org Assigned To
- Responsible Org Assigned To
- Administrative Org Assigned To Acceptance Indicator
- Responsible Org Assigned To Acceptance Indicator
- Defer Until Date
- Cause Code
- Trouble Found
- Cleared Code

These child ticket DMOQ attributes are updated only when closing ticket:

- TTR (Total Time to Repair)
- Total Customer Time
- Total Duration
- ETTR Provided Within 30 Mins of Ticket Open
- Service Restored Within 30 Minutes of ETTR
- Number Statuses Over 30 Minutes After Previous Status
- Number of Statuses Given
- Circuit In Service Date/Time

- Circuit In Service Within 30 Days of Ticket Open
- Circuit In Service Within 60 Days of Ticket Open

Updating Duplicate Tickets

The **duplicateTicketAllowed** Boolean field in the TicketImportInfo structure determines whether the API allows a change to the service item on a ticket if an open ticket already exists on the new service item, and both tickets have a ticket type that identifies repeat and chronic trouble.

About Customer Information and Updating Tickets

This section provides information about customer information and updating tickets.

Customer Must Be Passed as a Party ID

In the Trouble Management subsystem, users can optionally enter the Customer Name directly instead of selecting the customer from the Customer Search window. If the customer is not found, the Trouble Management subsystem displays an error when the ticket is saved. The updateTicket_v2 operation requires that the customer be passed in the form of a party ID if a customer is being specified. A client may still allow the user to enter the customer name directly and determine the ID by calling the getPartyByPartyName operation. That operation returns a party ID which can then be passed to the updateTicket_v2 operation.

Customer is Defaulted Based On the Service Item

If the service item is changed, and there is no customer on the ticket, the API defaults the customer to the customer associated in the MetaSolv Solution database with the service item. The customer billing address is also defaulted. The API writes a log note indicating that the customer was defaulted by the API.

Non-inventoried Service Items Are Not Created

A non-inventoried service item is not created if the service item on a ticket cannot be found in the MetaSolv Solution inventory. If the service item cannot be found, the new service item type is set on the ticket, but the service item description is set to null. A log note is created stating that the service item could not be found. The log note includes the service item identifier information passed in the ServiceItem structure.

Certain Codes are Passed as ID Values

Changes to the following codes on a ticket are passed in the form of their numeric ID values, not the code directly. Trouble Management API queries that return the numeric ID and the code are available for each. This enables you to populate dropdown fields on the client application.

- Escalation Method ID
- Initiating Mode ID
- Ticket Status ID
- Trouble Found ID
- Trouble Type ID

Ticket Dates and Times Are Exported and Imported in GMT

All dates exported by the getTicketForUpdate_v2 operation are exported in GMT. All dates imported in the updateTicket_v2 operation are assumed to be in GMT. It is the responsibility of the client to convert the exported dates to local time and the imported dates to GMT.

The export date and time returned by the getTicketForUpdate_v2 and getTicketReport operations are in the database server's time zone. The export date and time is passed back unchanged in the updateTicket operation and compared to the ticket's last modified date, which is stored in the database server's time zone.

Audit Note Date/Time Display

In the Trouble Management subsystem, the display of the date/time is determined by the setting on the client workstation and therefore varies depending on the user's individual settings. For the Trouble Management API, the standard format of **mm/dd/yyyy hh:mm:ss am/pm (GMT)** is used when giving details about date/time fields that have been updated, and these times are in GMT.

Telcordia Preference and Trouble Management API

MetaSolv Solution uses its Telcordia preference to determine if circuit identifier fields should be formatted according to Telcordia specifications (for example, having the proper number of spaces between virgules). The setting of the Telcordia preference has no effect when you use the Trouble Management API to specify a circuit for the createTicket_v2, createTicket_v3, updateTicket, or updateTicket_v2 operations, and the service item type is one of the types shown in Table 13–7. In such a case, the Trouble Management API searches the database for the value in the corresponding field as a formatted circuit. If the API does not find that value as a formatted circuit, the API searches again for that value as an unformatted circuit using the input provided in the operation's parameters.

Table 13–7 lists the service item types and the field names.

Table 13–7 Field Formatting

Service item type	Field name
Circuit/Connection	CircuitConnectionID
Internet Circuit	InternetCircuitIdent
Internet DSL	DSLCircuitIdent
Message Trunk Group	MsgTrunkGroupIdent

The clearTicket Operation

The clearTicket operation clears the designated ticket. That is, it changes the state of the ticket to 'Cleared'. This operation cannot be called on a ticket that is already in a cleared, closed or canceled ticket state. In addition, it cannot be called on a ticket that is an externally referred ticket state. You must first close (that is, verify) all the open external referrals through MetaSolv Solution.

A valid document number or ticket ID is required when a ticket is cleared. If both are passed, the ticket ID is ignored.

The following attributes are also required. These attributes may have already been set through the ticket update process prior to being cleared:

- Cause Code

- Trouble Found ID
- Cleared Code
- Ticket Status ID

Cleared Comment is an optional field.

The Trouble Found ID is the numeric ID associated with the trouble found code, and must be passed as a valid numeric value. The numeric ID values are returned with the trouble found codes in the getTroubleFoundCodes operation. The Trouble Found ID must be associated with the Cause Code as defined in the MetaSolv Solution infrastructure.

Likewise, the Ticket Status ID is the numeric ID associated with the ticket status code, and must be passed as a valid numeric value. The numeric ID values are returned with the ticket status codes in the getTicketStatusCodes2 operation. The Ticket Status ID must be associated with the Canceled ticket state as defined in the MetaSolv Solution infrastructure.

The above information is passed to the clearTicket operation in the UpdateableClearCloseInfo structure. Since this structure is also used by the closeTicket operation, it contains attributes for closing a ticket, including close contact first and last name, close contact access number, and close comment. These fields are ignored by the clearTicket operation. If you wish to clear and close a ticket at the same time, you can call the closeTicket operation.

You can also pass the clearTicket operation a sequence of log notes. These notes are displayed in the Clear Ticket event log entry along with the audit notes that are generated for each attribute that is changed. The log notes are not meant to replace the Cleared Comment.

Ticket Linkage and Clear Ticket

If the ticket being cleared is linked in a common cause relationship as a parent ticket, and the processChildTickets attribute is set to TRUE, the clearTicket operation automatically clears any child tickets that have not been cleared. You can determine whether the ticket is a parent or child by the ParentChildInd attribute returned by the getTicketForClearClose operation. The attribute is P if it is a parent, C if it is a child, and blank if it is not linked. If the ticket is a parent, the user should be prompt to ask if they wish to clear all child tickets with the parent.

All of the input information is applied to the child tickets that are cleared with the parent. The Cause Code, Trouble Found ID, and Cleared Code is applied to all child tickets that have not been closed or canceled, regardless of whether they are cleared with the parent.

If the ticket is a child ticket, the Cause Code, Trouble Found ID, and Cleared Code cannot be changed if they are already populated on the parent ticket.

Details Concerning Use of the closeTicket Operation

The closeTicket operation closes the designated ticket. That is, it changes the state of the ticket to 'Closed'. This operation cannot be called on a ticket that is already in a cleared, closed or canceled ticket state. In addition, it cannot be called on a ticket that is an externally referred ticket state. You must first close (i.e., verify) all the open external referrals through MetaSolv Solution.

A valid document number or ticket ID is required when a ticket is cleared. If both are passed, the ticket ID is ignored.

The following attributes are required, with the exception of Cleared Comment. These attributes may have already been set through the ticket update process prior to being cleared or when the ticket was cleared:

- Ticket Status ID
- Cause Code
- Trouble Found ID
- Cleared Code
- Cleared Comment

The Trouble Found ID is the numeric ID associated with the trouble found code, and must be passed as a valid numeric value. The numeric ID values are returned with the trouble found codes in the getTroubleFoundCodes operation. The Trouble Found ID must be associated with the Cause Code as defined in the MetaSolv Solution infrastructure.

Likewise, the Ticket Status ID is the numeric ID associated with the ticket status code, and must be passed as a valid numeric value. The numeric ID values are returned with the ticket status codes in the getTicketStatusCodes2 operation. The Ticket Status ID must be associated with the Canceled ticket state as defined in the MetaSolv Solution infrastructure.

With the exception of Close Contact Access Number, the following attributes are required when the state is changed to 'closed':

- Close Contact Last Name
- Close Contact First Name
- Close Contact Access Number
- Closed Comment

The API accepts a blank close contact first name or blank close contact last name, if one is provided.

If Close Contact Access Number is provided, it must be accompanied by the close contact first and/or last name. The close contact access number should be a telephone number, and can only contain numeric characters if the **Enable NPA/NXX Contact Telephone Number Formatting** preference is set to **Y** in MetaSolv Solution. This preference determines whether or not edit masks are used for contact phone numbers in MetaSolv Solution. If a telephone number is stored with formatting, it does not appear correctly when displayed in a field with an edit mask. If this preference is set to **Y** and the access number is not numeric, an error is returned.

You can also pass the closeTicket operation a sequence of log notes. These notes are displayed in the Close Ticket event log entry along with the audit notes that are generated for each attribute that is changed. The log notes are not meant to replace the Closed Comment.

Ticket Linkage and Close Ticket

If the ticket being closed is linked in a common cause relationship as a parent ticket, and the processChildTickets attribute is set to TRUE, the closeTicket operation automatically closes any child tickets that have not been closed or canceled. You can determine whether the ticket is a parent or child by the ParentChildInd attribute returned by the getTicketForClearClose operation. The attribute is P if it is a parent, C if it is a child, and blank if it is not linked. If the ticket is a parent, the user is prompted to confirm when closing all child tickets in the parent ticket.

All of the input information is applied to the child tickets that are closed with the parent. The Cause Code, Trouble Found ID, and Cleared Code is applied to all child tickets that have not been closed or canceled, regardless of whether they are closed with the parent.

If the ticket is a child ticket, the Cause Code, Trouble Found ID, and Cleared Code cannot be changed if they are already populated on the parent ticket.

Closing an Open/Active Trouble Ticket

As designed, the normal status life cycle of a trouble ticket proceeds from Open/Active at ticket creation, to Cleared when the trouble has been resolved but the customer has not yet verified that the service is restored, to Closed when the customer has verified that the service is working again.

Note: As designed, the optional states Externally Referred and Deferred are temporary diversions from that normal status life cycle. The optional Canceled state indicates a permanent closure of the ticket.

You can call the closeTicket operation on an Open/Active ticket, and the Trouble Management API can successfully clear and close the ticket at the same time. In this case, you must populate both the required fields for clearing a ticket and the required fields for closing the ticket. From the standpoint of both the Trouble Management subsystem and the Trouble Management API, there is no difference between calling closeTicket on an Open/Active ticket and calling clearTicket and closeTicket separately on the ticket.

Notifications for Cleared and Closed Tickets

When a ticket is cleared or closed through the Trouble Management API, the API sends a cleared or closed ticket notification to all escalation levels to which that ticket had been escalated, just as if the ticket had been cleared or closed in the Trouble Management subsystem. For tickets that were in the Cleared state when closed, the notification process is not called, because the notification process would have already been called when the ticket was cleared.

Details Concerning Use of the cancelTicket Operation

The cancelTicket operation cancels the designated ticket. That is, it changes the state of the ticket to 'Canceled'. This operation cannot be called on a ticket that is already closed or canceled.

A valid document number or ticket ID is required. If both are passed, the ticket ID is ignored.

With the exception of Close Contact Access Number, the following attributes are required when a ticket is canceled:

- Closed Comment
- Close Contact First Name
- Close Contact Last Name
- Close Contact Access Number
- Ticket Status ID

The API accepts a blank close contact first name or blank close contact last name, if one is provided.

If Close Contact Access Number is provided, it must be accompanied by the close contact first and/or last name. The close contact access number must be a telephone number, and can only contain numeric characters if the **Enable NPA/NXX Contact Telephone Number Formatting** preference is set to **Y** in MetaSolv Solution. This preference determines whether or not edit masks are used for contact phone numbers in MetaSolv Solution. If a telephone number is stored with formatting, it does not appear correctly when displayed in a field with an edit mask. If this preference is set to **Y** and the access number is not numeric, an error is returned.

The Ticket Status ID is the numeric ID associated with the ticket status code, and must be passed as a valid numeric value. The numeric ID values are returned with the ticket status codes in the getTicketStatusCodes2 operation. The Ticket Status ID must be associated with the Canceled ticket state as defined in the MetaSolv Solution infrastructure.

You can also pass the cancelTicket operation a sequence of log notes. These notes are displayed in the Cancel Ticket event log entry along with the audit notes that are generated for each attribute that is changed. The log notes are not meant to replace the Closed Comment.

Ticket Linkage and Cancel Ticket

If the ticket being canceled is linked in a common cause relationship as a parent ticket, the cancelTicket operation automatically cancels any child tickets that have not been closed. All of the input information is applied to the child tickets.

If the ticket being canceled is a child ticket, it is automatically unlinked from the parent ticket.

Details Concerning Use of the getTickets_v2 Operation

This operation enables you to query for a trouble ticket or a collection of tickets based on an optional set of criteria and a required sequence of ticket states. This functionality is similar to the MetaSolv Solution Ticket Search window.

The search criteria are passed in the form of a sequence of TicketQueryCriteria structures and a sequence of TicketStateEnum values. The TicketQueryCriteria structures are used to pass all of the searchable criteria that a ticket must meet to be returned by the query. The TicketStateEnum values are used to pass all of the possible ticket states that a ticket may be in to be returned by the query.

Note: When calling the getTickets_v2 operation, query criteria are optional. However, you must pass at least one ticket state, represented by a TicketStateEnum value, in the TicketStateQuerySeq structure.

If you pass no criteria but do pass a valid state, the response contains all trouble tickets of that state. However, you should exercise caution when doing so to avoid excessive processing time.

If you do not pass at least one valid state, the query fails regardless of the number of criteria you pass

---

The TicketQueryCriteria structure includes three attributes: **TicketSearchableField**, which is the field the criteria value must match against; **TicketSearchOperation**, which is the operator used in the search comparison, and a **string** value used to

compare against the searchable field. (See WDITroubleTypes_v3.idl for the enumerated values for **TicketSearchableFields** and **TicketSearchOperation** values).

Use the **maxRecords** parameter to limit the number of records the query returns. The operation limits the number of records returned to the lesser of the maxRecords parameter and the MetaSolv Solution Query Retrieval Limit preference. The operation returns the WDISearchResultsInfo structure which includes the limit used in the query and a Boolean indicating whether the matching records in the database exceeded the limit.

If no data is found given the query criteria, this operation returns the getTicketsNoDataFound_v2 operation on the Notification object.

The following rules apply to the search criteria:

1. The following searchable fields must be passed as numeric values:
   - InitiatingModeID
   - TicketStatusID
   - TroubleTypeID
   - TroubleFoundID
   - PriorityLevelID
   - SeverityLevelID
   - ResponsibleOrgPartyID
   - AdministrativeOrgPartyID
   - RespOrgAssingedToPartyID
   - AdminOrgAssignedToPartyID
2. The valid values for ServiceItemTypeCode include:
   - EQUIPMENT (used for Equipment)
   - CIRCUIT (used for Circuit)
   - MSG_TRNKGRP (used for Message Trunk Group)
   - EUS_TRUNKGRP (used for End User Special Trunk Group)
   - TELNBR (used for Telephone Number)
   - INTRNTDLP (used for Internet Dial-Up)
   - INTRNTCKT (used for Internet Circuit)
   - BWCKT (used Digital Subscriber Line)
3. The value passed for **ServiceItemDescription** is compared against the service item description on the trouble ticket. If **ServiceItemDescription** is passed, it must be accompanied by a **ServiceItemTypeCode**.
4. The valid values for **PriorityLevelID** must be 0, 1, 2, or 3, where these values have the following definitions:
   - 0 - Undefined
   - 1 - Minor
   - 2 - Major
   - 3 - Serious

5. The valid values for **SeverityLevelID** must be 0, 1, 2, or 3, where these values have the following definitions:
   - 0 - Out of Service
   - 1 - Back in Service
   - 2 - Service Impairment
   - 3 - Non Service Affecting Trouble
6. The **DateRangeType** identifies which date field to apply the DateRangeFromDate and DateRangeToDate criteria to. Valid values include:
   - OPEN DATE
   - TROUBLE DETECTION DATE
   - CLEARED DATE
   - CLOSE DATE
7. If **DateRangeFromDate** and **DateRangeToDate** criteria are passed, then **DateRangeType** must also be passed.
8. If either **DateRangeFromDate** or **DateRangeToDate** criteria are passed, both must be passed.
9. The values for **DateRangeFromDate** and **DateRangeToDate** must be valid dates and time passed **int** the format of "YYYYMMDDHHMMSS" with the hours in 24-hour notation, sometimes referred to as military time. The date and time values are expected to be passed in the GMT time zone. The previous *getTickets* version expected only the date portion.
10. If passed, the **DateRangeToDate** cannot be a date and time prior to **DateRangeFromDate**.
11. The **TicketSearchableField** values listed below may be used with all of the TicketSearchOperation operators. All other ticket searchable fields may only be used with the EQUAL operator.
    - DocumentNumber
    - TicketID
    - CustTroubleTicketNum (Customer Trouble Ticket Number)
    - ServiceItemDescription
    - CustomerName
    - ExtRefTicketNum (External Referral Ticket Number)

The causeCode parameter limits the trouble found codes that are returned to only those that are related to this cause code. If activeOnly is passed as true, the cause code is required and must be a valid active or inactive cause code in the Trouble Management subsystem.

Details Concerning Use of the Service Item Query Operations

The Trouble Management subsystem provides a Service Item query window which may be accessed when editing a trouble ticket in order to find a service item to associate to the ticket. This window presents a different query for each service item type. Query operations that provide similar functionality are available in the Trouble Management API and ICM API. These queries can be used to retrieve the service item identifier that is passed to the createTicket_v3 and updateTicket_v2 operations.

The service item queries for the Circuit and Equipment service item types are not located in the Trouble Management API. To query for circuits, you may use the getQueryCircuits operation located in the ICM API (WDIEquipment.idl). To query for equipment, use the queryEquipInstall_v2 operation, also located in the ICM API (WDIEquipment.idl). For more information about these queries, see "The Inventory and Capacity Management API".

The service item query operations for the remaining service item types, are located in the Trouble Management API. These operations include:

- getMsgTrnkGrpServItem (message trunk groups)
- getEUSpecialTrnkGrpServItem (end user special trunk groups)
- getTelephoneNumberServItem (telephone numbers)
- getInternetCircuitServItem (Internet circuits)
- getInternetDialupServItem (Internet dial-ups)
- getDSLServItem (digital subscriber lines)

In each of these operations in the Trouble Management API, with the exception of getTelephoneNumberServItem, the criteria are passed as a sequence of structures, where the structure includes:

- An enumerated field indicating the field to be searched. The enumerated values for this field correspond to the criteria fields on the query window in the Trouble Management subsystem. Only one structure may be passed for a given searchable field.
- An enumerated field representing the operator used in the search comparison. For example, EQUAL or LIKE.
- A string value which is compared against the field being searched

Each service item query operation in the Trouble Management API is also passed a maxRecords parameter to limit the number of records the query returns. The operation limits the number of records returned to the lesser of the maxRecords parameter and the Query Retrieval Limit preference in MetaSolv Solution. Along with the query results, each operation returns the WDISearchResultsInfo structure which includes the limit used in the query and a Boolean indicating whether the matching records in the database exceeded the limit.

If no data is found given the query criteria, these operations return a “NoDataFound” operation on the Notification object. For example, getDSLServItemNoDataFound.

Structure Format Criteria for the getTelephoneNumberServItem Operation

To use the getTelephoneNumberServItem operation, you must identify the structure format that applies to the telephone number(s) you are searching for. The structure format defines the components that make up a telephone number. The criteria input for this operation is passed in the StructureFormat structure, which consists of the following attributes:

- Structure Type: For this operation, the type must be set to **TN** (Telephone Number)
- Structure Name: This identifies the structure format that applies to the telephone number(s) for which you are searching. For example, TN-US. You can call the getStructureFormatsGivenType operation in the Infrastructure API to get a list of valid telephone number structure formats.

- Components: This is a sequence component structures you want to include as criteria. You can get the component details needed to fill out this structure by calling the getComponentsGivenStructureFormat operation in the Infrastructure API with the structure name. The following attributes are included in the `SFComponent` structure:
  - Component ID: This is an internal unique identifier for the component.
  - Component Name: This is name of the telephone number component. For example, NPA.
  - Component Type: This is a categorization of the component. For example, "T" (table driven).
  - Component Value: This is the component criteria value to be used in the search.

The component ID, name and type must be valid for the structure name, or an error is returned. Only one SFComponent structure may be passed for a given component. A structure format may contain components that are required as criteria in a search. These components can be identified through the output of the getComponentsGivenStructureFormat operation in the Infrastructure API. The **requiredIndicator** attribute equals **Y** if the component must be included as criteria.

MetaSolv Solution Software Concepts

This section describes important concepts about the MetaSolv Solution software.

Overview of the Trouble Management Subsystem

Successful development using the Trouble Management API requires an understanding of the Trouble Management subsystem.

The Trouble Management subsystem tracks a reported problem from its initial identification to its resolution. The Trouble Management subsystem maintains information such as contacts, trouble codes, cause codes, and priority and escalation levels. The Trouble Management subsystem records the information necessary to allow the creation of various reports, including DMOQ reports for trouble tickets.

Trouble tickets can be associated with existing circuits, telephone numbers, trunk groups, and customers. The Trouble Management subsystem tracks all activities associated with resolving a ticket and monitors the ticket's state, status and responsible and administrative organizations throughout the ticket's life cycle.

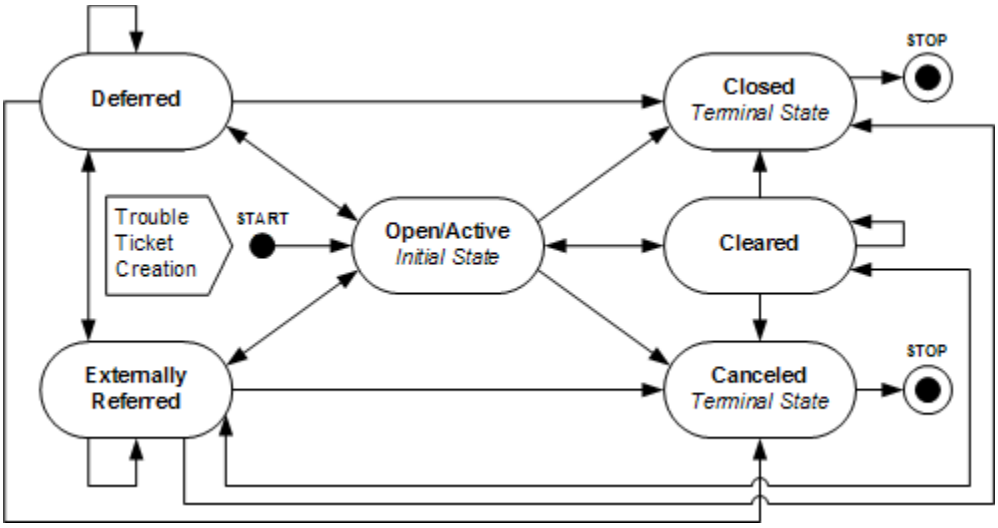
See the online Help for more information.

Permitted Trouble Ticket State Changes

MetaSolv Solution users can change the ticket state of trouble tickets through menu selections within the Trouble Management subsystem.

Figure 13–2 illustrates the ticket state changes permitted by the Trouble Management subsystem.

Figure 13–2 Ticket State Changes Permitted By The Trouble Management Subsystem



The rules that govern ticket state changes through the Trouble Management subsystem are listed below:

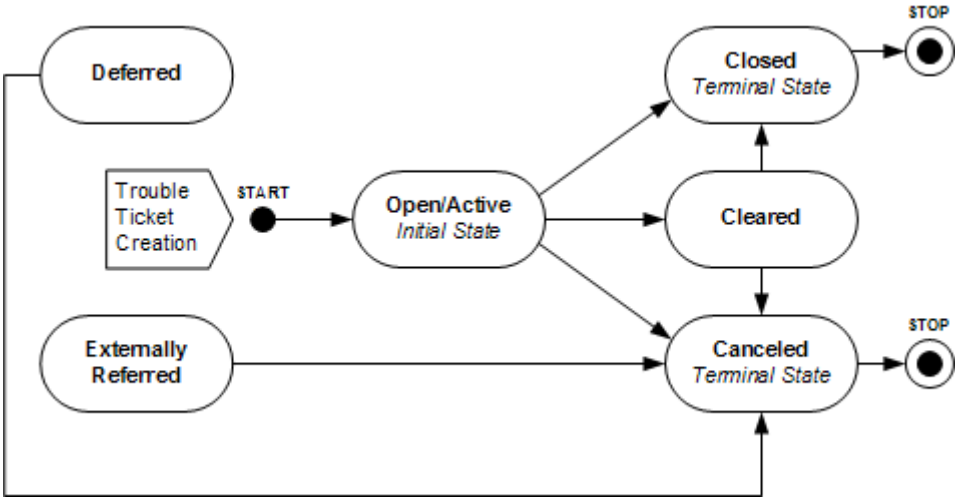
- Tickets are created in the Open/Active state.
- Open Active tickets can be changed to any ticket state except Open Active.
- Deferred tickets can be changed to any ticket state except Cleared.
- Externally Referred tickets can be changed to any ticket state.
- Cleared tickets can be changed to any ticket state except Deferred.
- Closed and Canceled tickets cannot change state.

> **Note:** Closed and canceled are terminal states. Once a trouble ticket is placed in one of these states, the ticket state can never be changed again. Non-reporting details for a closed or canceled trouble ticket can be added or edited. Reportable details cannot be added or edited, except as permitted by the setting of the Allow Editing of Task Completion Date Within the Grace Period preference.

The Trouble Management API enables you to use the updateTicket_v2 operation to change trouble ticket statuses, but only for a subset of the status changes possible through MetaSolv Solution.

Figure 13–3 illustrates the ticket state changes permitted by the Trouble Management API.

Figure 13–3 Ticket State Changes Permitted by The Trouble Management API



The rules that govern ticket state changes through the Trouble Management API are listed below:

- Tickets are created in the Open/Active state using the createTicket_v3 operation.
- Open Active tickets can be changed to Cleared, Closed, and Canceled.
- Deferred and Externally Referred tickets an only be changed to Canceled.
- Cleared tickets can be changed to Closed and Canceled.
- Closed and Canceled tickets cannot change state The updateTicket operation does not permit changes to closed or canceled tickets.

Trouble Management Operational Differences

This section provides information about the operational differences between the Trouble Management subsystem and the Trouble Management API.

Escalation Organizations and Levels and the Trouble Management API

The Trouble Management API does not accept import of the information that fills these editable fields on the Escalations tab of the Trouble Management subsystem's New Ticket window. The Trouble Management API defaults these values when a trouble ticket is created:

- Admin Org to Notify - Level
- Admin Org to Notify - Notify Ind
- Other Org to Notify - Level
- Other Org to Notify - Notify Ind
- Other Orgs to Notify - Org
- Resp Org to Notify - Level
- Resp Org to Notify - Notify Ind

The Trouble Management API defaults these values when a ticket is created or when the **Administrative Organization**, **Responsible Organization**, or **Escalation Method** is updated on a ticket.

External Referrals and the Trouble Management API

The Trouble Management subsystem allows a ticket to be externally referred to multiple maintenance center organizations.

The Trouble Management API does not support the creation or maintenance of external referrals. If a ticket has been externally referred, it may be updated using the updateTicket_v2 operation, and it may be canceled using the cancelTicket operation. However, an externally referred ticket cannot be cleared or closed through the Trouble Management API.

User-required Optional Trouble Management Subsystem Fields and the Trouble Management API

The Trouble Management subsystem allows users to require entries for fields that the Trouble Management subsystem defines as optional.

The Trouble Management API does not enforce user-defined requirement of optional Trouble Management subsystem fields when you submit a trouble ticket through the Trouble Management API. Instead, the Trouble Management subsystem requires the first user who updates that trouble ticket in the Trouble Management subsystem to enter the information for the user-required Trouble Management subsystem fields.

User-defined Fields and the Trouble Management API

The Trouble Management subsystem allows users to create user-defined fields and to require entries in those fields on a trouble ticket.

The Trouble Management API does not support import of information for user-defined fields, and user-defined fields are not returned by the getTicketReport_v2 operation.

If a user-defined field is required, the Trouble Management subsystem requires the first user who updates that trouble ticket in the Trouble Management subsystem to enter the information for the field.

Certain Field Values Not Defaulted

In the Trouble Management subsystem, the values of the **Customer Status Minutes** field and the corresponding **Next Customer Status Date and Time** field are available in the IDL for the Trouble Management API, which permits your client application to set these values as required.

When you use the updateTicket _v2 operation to change the **Customer Status Minutes** field, the updateTicket _v2 operation does not automatically calculate and set the **Next Customer Status Date/Time** field. Likewise, when you use the updateTicket _v2 operation to clear the **Customer Status Minutes** field, the updateTicket_v2 operation does not automatically clear the **Next Customer Status Date/Time** field.

No Default of ETTR, Priority Level or Customer Status Minutes for a Circuit Service Item

In the Trouble Management subsystem, users can set up defaults for the **Estimated Time To Restore** (**ETTR**), **Priority Level** or **Customer Status Minutes** fields, based on the service type code, service type category, and trouble type for a circuit service item. The updateTicket_v2 operation does not default these fields when an inventoried circuit is set on a ticket.

Repeat and Chronic Trouble Ticket Types

Trouble tickets can either represent real service issues, such as service outages and equipment failures, or can be informational in nature. Whether a given trouble ticket is informational or represents a real service issue is determined by the setting of the **Identifies Repeat and Chronic Trouble Indicator** check box on the Trouble Management subsystem's Ticket Type window for the ticket's trouble ticket type.

- If the **Identifies Repeat and Chronic Trouble Indicator** check box is selected, the trouble ticket represents a real service issue.
- If the **Identifies Repeat and Chronic Trouble Indicator** check box is deselected, the trouble ticket is informational.

Whenever a new trouble ticket is entered into the database, whether through the Trouble Management subsystem or the Trouble Management API, the ticket is evaluated to determine whether it could constitute an instance of repeat trouble, chronic trouble, or both. In order for a trouble ticket to represent an instance of repeat or chronic trouble, the trouble ticket type's **Identifies Repeat and Chronic Trouble Indicator** check box must be checked, and the appropriate condition below must be met:

- For repeat trouble, at a minimum one trouble ticket that represents a real service issue must have been entered for that service item within the past 30, 60, 90, or greater than 90 days.
- For chronic trouble, the service item must have had a minimum number of trouble tickets that represent real service issues within a maximum number of days in the past. These minimum and maximum values are determined by the setting of the Trouble Management subsystem's Chronic Trouble Number of Tickets and Number of Days preference.

  A given service item can have multiple informational trouble tickets in an open ticket state at the same time. However, while a given service item has a trouble ticket that represents a real service issue that is in a ticket state other than *Closed* or *Canceled*:

- The Trouble Management API cannot accept additional trouble tickets that represent a real service issue for that service item.
- The Trouble Management subsystem warns users who enter a trouble ticket that represents a real service issue for that service item that the open ticket exists and asks the user if they want to create the new ticket anyway.

Effect of Data Errors in Trouble Reports on Trouble Management API Processing

If your application submits, to the Trouble Management API, a trouble ticket that omits non-critical information or has a non-critical error, the Trouble Management API creates the trouble ticket and adds log notes to the trouble ticket that identify the missing information or data errors. For example, a contact phone number was given, but no contact name was supplied.

If your application submits to the Trouble Management API a trouble ticket that omits critical information or has a critical error, the Trouble Management API rejects the trouble report with an explanation, for example the ticket type code that is passed does not exist in the database.

14

The Work Management API

The Work Management API exposes certain functions of the Oracle Communications MetaSolv Solution Work Management subsystem and certain information in the database that the Work Management subsystem uses.

- Implementation of external applications that use the Work Management API follows the pattern described in "Asynchronous Interaction Pattern".

The Work Management API can be used to provide limited access to Work Management subsystem functions and information from remote and local locations for both field personnel and other users of MetaSolv Solution. Possible examples of applications that can be developed using the Work Management API are:

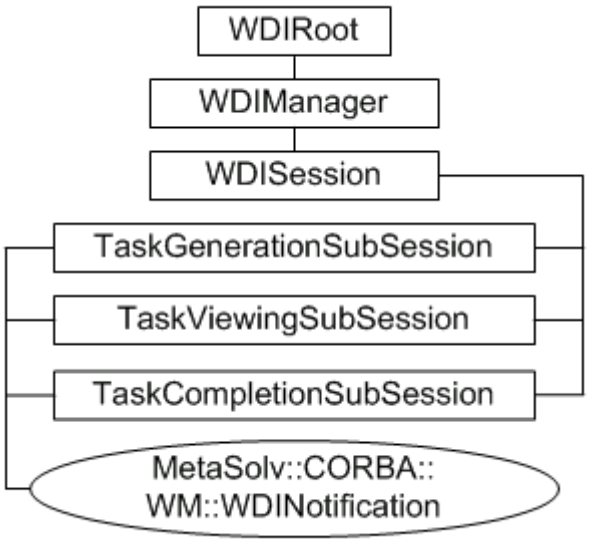
- An application that electronically generates tasks for service requests that are received electronically, which eliminates the need to generate tasks for these service requests manually.
- A Web interface or application that monitors a work queue, reports new tasks in that queue to the user (an individual or work group), and reports completion of specific tasks and gateway events by that user back to the Work Management API. For example, if the credit department must complete a credit check task prior to order completion, the credit department could use an application that notifies them when credit check tasks have been assigned to them. As the assigned tasks are completed, the credit department could use this application to report completion of the tasks.
- A thin client or Web interface for field personnel that displays their work queue, displays the service request for which the task is performed, displays the relationships and dependencies between tasks, and allows users to report task completion and the reason that tasks were completed late. This type of application could be used in situations where it is difficult or impossible to run the entire MetaSolv Solution application remotely.

The CORBA servername used by the Work Management API is WMSERVER.

WMSession Interfaces

Figure 14–1 shows the relationship of the interfaces within the Work Management API.

Figure 14–1 WMSession Interfaces



WDIManager

Table 14–1 lists the operations available in the WDIManager interface of the **WDIWM.IDL** file.

Table 14–1 WDIManager Interface Operations in Work Management API

Operation	Description
startWMSession	Obtains the object reference of the WMSession
destroyWMSession	Terminates the WMSession
startTransaction	commit rollback
destroyTransaction	Terminates the transaction
startSignal	eventOccurred eventTerminated eventInProgress eventCompleted eventErrored
destroySignal	Terminates the signal
startInSignal	eventInProgress eventCompleted eventErrored
destroyInSignal	Terminates the InSignal

Note: See "Common Architecture" for more information on the WDIManager interface.

WMSession Interfaces

Table 14–2 lists the three operations that comprise the WMSession in the **WDIWM.IDL** file.

Table 14–2 Work Management API WMSession Interface Operations

Operation	Description
startTaskGenerationSubSession	Obtains the object reference for the TaskGenerationSubSession
destroyTaskGenerationSubSession	Triggers destruction of the TaskGenerationSubSession object
startTaskViewingSubSession	Obtains the object reference for the TaskViewingSubSession
destroyTaskViewingSubSession	Triggers destruction of the TaskViewingSubSession object
startTaskCompletionSubSession	Obtains the object reference for the TaskCompletionSubSession
destroyTaskCompletionSubSession	Triggers destruction of the TaskCompletionSubSession object

WMSession Interface Operation Descriptions

- startTaskGenerationSubSession

  Obtains the object reference for the TaskGenerationSubSession

- destroyTaskGenerationSubSession

  Triggers destruction of the TaskGenerationSubSession object

- startTaskViewingSubSession

  Obtains the object reference for the TaskViewingSubSession

- destroyTaskViewingSubSession

  Triggers destruction of the TaskViewingSubSession object

- startTaskCompletionSubSession

  Obtains the object reference for the TaskCompletionSubSession

- destroyTaskCompletionSubSession

  Triggers destruction of the TaskCompletionSubSession object

The *requestID* parameter used by many of the operations in the Work Management API is an arbitrary, user-defined number that provides a means of relating requests and notifications when performing asynchronous operations. The Work Management operations do not make use of this parameter. Instead, they return it unchanged and unevaluated when executing the notification method.

Many of the descriptions of the operations in the Work Management API state that the operation returns a value or values. In such cases, remember that the operation returns that value by invoking the appropriate response operation on the notification object.

TaskGenerationSubSession Interfaces

Table 14–3 lists the operations available in the TaskGenerationSubSession session of the **WDIWM.IDL** file.

Table 14–3 Work Management API TaskGenerationSubSession Interface Operations

Operation	WDINotification Operations
generateAndSaveTasks	generateAndSaveTaskSucceeded generateAndSaveTaskFailed
getAllQueues	getAllQueuesSucceeded getAllQueuesFailed
getAllProvPlans	getAllProvPlansSucceeded getAllProvPlansFailed
getPlanID	getPlanIDSucceeded getPlanIDFailed
getAutoPlanID	getAutoPlanIDSucceeded getAutoPlanIDFailed

TaskGenerationSubSession Interface Operation Descriptions

- generateAndSaveTasks

  Given an order (document_number), a provisioning plan ID, and the time zone of the client, this operation generates tasks for that order along with completion dates for each task. A sequence of tasks with their dates are returned along with a sequence that contains the relationship between these tasks and a status. generateAndSaveTasks supports the MetaSolv Solution rules and behaviors functionality when generating tasks.

- getAllQueues

  This operation provides the functionality to return a sequence of all available work queues in the MetaSolv Solution database if the work queues are to be manually assigned.

- getAllProvPlans

  This operation provides the functionality to return a sequence of all available provisioning plans in the MetaSolv Solution database if the provisioning plan is to be assigned manually.

- getPlanID

  This operation provides the functionality to return a specific provisioning plan name specified by the third party developer. This operation returns the ID of the plan using a plan name. This is an alternative method to choosing a default provisioning plan for internet services.

- getAutoPlanID

  This operation provides the functionality to automatically pick a provisioning plan based on predefined third-party criteria. This operation returns the first plan ID which is defined under the organization, jurisdiction, and the service group of the given order (document_number.)

TaskViewingSubSession Interface Operations

Table 14–4 lists the operations available in the TaskViewingSubSession.

Table 14–4 TaskViewingSubSession Interface Operations

Operation	WDINotification Operations
getUserWorkQueue	getUserWorkQueueSucceeded getUserWorkQueueFailed
getWorkGroupWorkQueue	getWorkGroupWorkQueueSucceeded getWorkGroupWorkQueueFailed
getTasks	getTasksSucceeded getTasksFailed
getPredecessorTasks	getPredecessorTasksSucceeded getPredecessorTasksFailed
getFollowerTasks	getFollowerTasksSucceeded getFollowerTasksFailed
getTaskCircuits	getTaskCircuitsSucceeded getTaskCircuitsFailed
getTaskChecklist	getTaskChecklistSucceeded getTaskChecklistFailed
getTaskGWEvent	getTaskGWEventSucceeded getTaskGWEventFailed
updateChecklist	updateChecklistSucceeded updateChecklistFailed
updateGWEvent	updateGWEventSucceeded updateGWEventFailed
getServReqTasks	getServReqTasksFailed getServReqTasksSucceeded
acceptTask	acceptTaskFailed acceptTaskSucceeded
updateEstCompDate	updateEstCompDateFailed updateEstCompDateSucceeded
transferTask	transferTaskFailed transferTaskSucceeded
rejectTask	rejectTaskFailed rejectTaskSucceeded
searchWorkQueue	searchWorkQueueFailed searchWorkQueueSucceeded
getTaskDetail	getTaskDetailFailed getTaskDetailSucceeded
getServReqDetail	getServReqDetailFailed getServReqDetailSucceeded
getServReqNotes	getServReqNotesFailed getServReqNotesSucceeded

Table 14–4 (Cont.) TaskViewingSubSession Interface Operations

Operation	WDINotification Operations
addServReqNote	addServReqNoteFailed addServReqNoteSucceeded
getTaskJeopardy	getTaskJeopardyFailed getTaskJeopardySucceeded
getTaskJeopardyDetail	getTaskJeopardyDetailFailed getTaskJeopardyDetailSucceeded
addTaskJeopardy	addTaskJeopardyFailed addTaskJeopardySucceeded
updateTaskJeopardy	updateTaskJeopardyFailed updateTaskJeopardySucceeded
deleteTaskJeopardy	deleteTaskJeopardyFailed deleteTaskJeopardySucceeded
getJeopardyCode	getJeopardyCodeFailed getJeopardyCodeSucceeded

TaskViewingSubSession Interface Operation Descriptions

- getUserWorkQueue

  This operation provides the functionality to return all work queues owned by the user ID passed in to the operation. This process uses some of the existing functionality and SQL used in the Work Management subsystem to build a list of personal work queues.

- getWorkGroupWorkQueue

  This operation provides the functionality to return all work queues except those owned by the user ID passed in to the operation. This process uses some of the existing functionality and SQL used in the Work Management subsystem to build a list of work queues.

- getTasks

  This operation provides the functionality to return task information for the work queue passed in to the operation. Date/time fields are converted to local time using the local time zone that is passed in. Task information returned includes task_type, task_status, revised_completion_date, queue_status, type_of_sr (type of service request) pon, first_ecckt_id, document_number, and task_number.

- getPredecessorTasks

  This operation provides the functionality to return the task information of predecessor tasks for a given task. Predecessor task information includes task_type, task_status, scheduled_completion_date, actual_release_date, revised_completion_date, estimated_completion_date, work_queue_id, actual_completion_date, task_critical_date_ind (critical task ind), task_status_date, document_number, task_number, first_jeopardy_id (jeopardy ind), and auto_comp_ind (auto completion ind.)

- getFollowerTasks

  This operation provides the functionality to return follower task information for a given task. Follower task information includes task_type, task_status, scheduled_

completion_date, actual_release_date, revised_completion_date, estimated_ completion_date, work_queue_id, actual_completion_date, task_critical_date_ind (critical task ind), task_status_date, document_number, task_number, first_ jeopardy_id (jeopardy ind), and auto_comp_ind (auto completion ind.)

- getTaskCircuits

  This operation provides the functionality to return circuit information as it relates to a given task. Task circuit information includes ecckt (circuit ID), act_comp_date (circuit completion date), jeopardy_ind, ckt_design_id, complete_ind (circuit completion ind) and, notes_ind (circuit notes ind.)

- getTaskChecklist

  This operation provides the functionality to return the checklist items for a given task. Task checklist information includes check_code (checklist identifier code), check_comp_date (checklist completion date), check_seq, check_desc (checklist description), and complete_ind (checklist completion ind.)

- getTaskGWEvent

  This operation provides the functionality to return the gateway events for a given work_queue_id. Task gateway event information includes event_id, event_nm (event name), task_type, task_type_pre (predecessor task to gateway event's task), force_reopen_ind, status_cd, version, signal_ind, in_out_cd, event_detail, document_number, task_number, task_number_pre, and serv_item_id.

- updateChecklist

  This operation updates the MetaSolv Solution database when the user changes the **Completion Indicator** field. The completion date for the checklist item is set to null if the completion indicator is set to N or it is set to the current date and time if the completion indicator is set to Y.

- updateGWEvent

  This operation provides the functionality to update the **Status** field in the gateway events tables.

- getServReqTasks

  This operation returns task information for a given document number (service request). This operation is oriented more toward the view of the service request than the *getTasks* method is. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- acceptTask

  In the Work Management subsystem, users acknowledge tasks that have been placed in their work queue by accepting them. This operation enables you to acknowledge a task that has been placed in a work queue.

- updateEstCompDate

  This operation updates the estimated completion date for a specified task. Date and time information is stored in the database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- transferTask

This operation transfers the specified task from the work queue identified by the current WorkQueue parameter to work queue identified by the newWorkQueue parameter.

- rejectTask

  This operation rejects a completed predecessor task. You reject a task to return it to the work queue of the person who completed that task so they can rework the task.

  When you use the rejectTask method, the Work Management API changes the rejected task's reject status to R (Rejected) and the task's status to Ready. The API also changes the reject status of all completed follower tasks to R and sets their status to Pending.

  > **Note:** You can find a task's reject status in the **rejectStatus** field in the predFollow structure and the **taskRejectStatus** field in the taskView structure.
  >
  > You can find a task's status in the **taskStatus** field in the predFollow structure and the **taskStatus** field in the taskView structure.

- searchWorkQueue

  This operation takes a string or partial string passed in through the searchKey parameter and tries to match it to existing work queues in the MetaSolv Solution database. The operation returns a sequence of all work queues that match the search criteria. The type of search is determined by the searchType parameter, a value of "B" requests a "Begins with" search, and a value of "C" requests a "Contains" search.

- getTaskDetail

  This operation returns task detail information for a given document number and task number. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- getServReqDetail

  This operation returns basic service request information for a given document number. Date/time information is returned using the timezone you specify in the timezone parameter. Service request detail information includes type of service request, service request status, responsible party, purchase order number, order number, desired due date, supplement type, and CCNA.

- getServReqNotes

  This operation returns a sequence of all notes that have been entered for the designated service request.

- addServReqNote

  This operation adds a service request note for the designated service request.

- getTaskJeopardy

  This operation returns a sequence of task jeopardy information for the document number/task number passed in to the method. Jeopardy information is used to

identify why a task is or was in jeopardy of being completed late. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- getTaskJeopardyDetail

  This operation returns task jeopardy information for a single jeopardy ID passed in to the method. Jeopardy information is used to identify why a task is or was in jeopardy of being completed late. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- addTaskJeopardy

  This operation adds task jeopardy information for the task number you designate. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- updateTaskJeopardy

  This operation updates task jeopardy information for the task number you designate. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks that will be performed in different time zones.

- deleteTaskJeopardy

  This operation deletes jeopardy information for a given task on a given service request.

- getJeopardyCode

  This operation returns a sequence of all available jeopardy codes in the MetaSolv Solution database.

TaskCompletionSubSession Interface Operations

Table 14–5 lists the operations available in the TaskCompletionSubSession.

Table 14–5 TaskCompletionSubSession Interface Operations

Operation	WDINotification Operations
getOrganization	getOrganizationSucceeded getOrganizationFailed
getWhyMissCode	getWhyMissCodeSucceeded getWhyMissCodeFailed
completeTask	completeTaskSucceeded completeTaskFailed

Table 14–5 (Cont.) TaskCompletionSubSession Interface Operations

Operation	WDINotification Operations
completeTaskOnDate	completeTaskSucceeded completeTaskFailed
reopenTask	reopenTaskFailed reopenTaskSucceeded
validateEditActCompDate	validateEditActCompDateFailed validateEditActCompDateSucceeded
searchCompletedTasks	searchCompletedTasksFailed searchCompletedTasksSucceeded

TaskCompletionSubSession Interface Operation Descriptions

- getOrganization

  This operation returns a sequence of all available organization IDs for the organization type defined in the MetaSolv Solution **Jeopardy Code Organization Type** preference.

- getWhyMissCode

  This operation provides the functionality to return a sequence of whymissed codes used when selecting a whymissed code in the task completion process.

- completeTask

  Given a document number and task number, this operation validates the task to ensure it is ready to be completed. If it passes validation, and is on time, the task is completed. If the task is being completed late, a whymissed code is assigned before completing the task.

- completeTaskOnDate

  This operation completes the task represented by the passed document number and task number, and sets the revised completion date to the passed completionDate if the following conditions are true: The task is late, but not beyond its grace period, and the **Allow Edit of Task Completion within Grace Period** preference is set to Y. Otherwise, the passed completionDate is ignored.

  Note: The completeTaskOnDate operation uses the same return codes as the completeTask operation.

- validateEditActCompDate

  This operation validates whether or not a task's actual completion date can be edited.

- reopenTask

  Reopens the completed task that you identify by document number and task number.

WARNING: Reopening tasks is not recommended for the following reasons:

- **There is no notification, to the owner of a queue, that a task is a reopened task, and no indication in the status column that the status is "Reopened".**
- **If the reopened task has any associated gateway events, those gateway events must be reactivated.**
- **If the reopened task is a precondition for a gateway event on another task, that gateway event must be reactivated.**
- **If there are any completed follower tasks to the one you want to reopen, you must reopen the follower tasks first. If the follower tasks are not in your own work queue, they reappear, when reopened, in their original work queues with a status of Pending. The original queue's owner does not receive notification of reopened tasks in their queue.**

- searchCompletedTasks

  This operation returns a sequence of completed tasks that meet the passed search criteria. Date and time information is stored in the MetaSolv Solution database in Greenwich Mean Time (GMT). The Work Management API converts dates and times between the time zone identified by the timezone parameter and the GMT equivalent. This allows coordination of tasks performed in different time zones.

Work Management API IDL Files

The following IDL files are included in the Work Management API:

- WDIWM.IDL
- WDIWMTYPES.IDL
- WDIWMTYPES_V2.IDL

Process Flows

This section contains a sample process flow for a solicited message. Use the sample flows as templates for developing your own process flows.

See "Unsolicited Messages" for the process flow that is used when the Work Management API is the client.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. The API plays the role of the client, and the third party application plays the role of the server.

Table 14–6 lists the interfaces and operations that the third-party application implements using the IDL files provided with the Work Management API.

Table 14–6 Work Management API Solicited Message Operations

Interface	For Implementing These Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	N/A
WDISignal	eventOccurred eventTerminated
WDIInSignal	N/A

Sample Solicited Message Process Flow

When the Work Management API is the client, the overall process flows as follows:

1. The API client binds to the third-party server to get a WDIRoot object reference.
2. The API client invokes the *connect* operation of the WDIRoot interface, which yields a WDIManager object reference.
3. The API client invokes the *startSignal* operation of the WDIManager interface to get a WDISignal object reference.
4. The API client invokes the *eventOccurred* operation of the WDISignal interface passing a WDIEvent structure to notify the third-party application that an event registered to them has occurred within the database.
5. The API client invokes the *destroySignal* operation of the WDIManager interface.
6. The API client invokes the *disconnect* operation of the WDIRoot interface.
7. Once the third-party server completes processing, possibly involving additional unsolicited messages to the MetaSolv Solution Application Server, the third party server binds to the application server and follows the same process described above for the MetaSolv Solution client with the exception that the *eventCompleted/Errored* operations are invoked passing the original WDIEvent structure.

If the third-party application encounters an error, it throws a WDIExcp as defined by the IDL. The client handles CORBA system exceptions and WDIExcp exceptions.

Unsolicited Messages

An unsolicited message is a message initiated by the third-party application. For an unsolicited message, the Work Management API plays the role of the server and the third-party application plays the role of the server with the exception of callback processing.

See "Solicited Messages" for the process flow used when the Work Management API is the client.

Enhanced Off-net Automation Functionality and the Work Management API

Table 14–7 lists the interfaces and operations that the Work Management API server (DLRSERVER) implements using the IDL files provided with the Work Management API.

Table 14–7 Work Management API Unsolicited Message Operations

Interface	For Implementing Only These Operations
WDIRoot	connect disconnect
WDIManager	startTransaction destroyTransaction
WDITransaction	commit rollback
WMSession	getWMSession

The Work Management subsystem provides enhanced automation of off-net orders, as follows:

- Provisioning plan templates can define relationships between tasks on PSRs and tasks on child LSRs. At task generation, when the defined conditions exist for the orders, the Work Management subsystem automatically creates the relationships between the tasks.
- During the CONF or RCONF task, the due date for the DD task and due dates for predecessor or child tasks can be adjusted automatically based on the FOC date received from an external provider of an LSR or ASR child order.

The Work Management API supports the operation of both of these features. When tasks are generated through the Work Management API, and the defined conditions exist for the orders, the Work Management subsystem automatically creates the relationships between the tasks. When you use the Work Management API to complete a CONF task, and the appropriate conditions exist, the Work Management API automatically adjusts the Due Dates for the order and its tasks as appropriate.

Before you can use these automated features of MetaSolv Solution, you must use MetaSolv Solution to set up the relationships between the provisioning plans and to identify for each ICSC whether automatic date adjustment is permitted. See the online Help for more information.

Implementation Concepts

This section describes the concepts that enable you to implement the Work Management API.

Overview of the MetaSolv Solution Work Management Subsystem

To successfully develop applications using the Work Management API, you must first have a thorough understanding of the MetaSolv Solution Work Management subsystem.

The Work Management subsystem provides users with the tools needed to complete these activities:

- Define a variety of provisioning plans, which are generic templates of tasks needed to fulfill specific types of service requests
- Define rules under which MetaSolv Solution can dynamically change a provisioning plan at the time it applies the plan to a specific service request
- Apply a provisioning plan to a specific service request and:

- Generate and modify the specific set of tasks needed to fulfill a service request
- Define and modify the dependency relationships between tasks
- Define and modify the due dates for tasks
- Electronically schedule and assign tasks to individuals and work groups such as departments and field offices across the organization
- Track and report on task completion
- Report why a task was completed late

After each service request is entered, a user generates the tasks for that service request. During task generation, the user can add or remove tasks, change task due dates, adjust the dependency relationships between tasks, and determine the work queue to which each task is assigned.

After any needed adjustments are made to the tasks, and the work queues are selected, the Work Management subsystem dispatches the tasks to the specific work queues.

In the Work Management subsystem, after a given task is completed:

- The completed task's predecessor tasks and immediate follower tasks can no longer be added to or removed from the service request
- The dependency relationships between the completed task and its predecessor tasks and immediate follower tasks can no longer be changed

After a worker is assigned to a task, the worker can use the Work Management subsystem to monitor the status of the tasks assigned to them. When a task reaches Ready status, the worker performs the work and changes the status of the task to Completed. The Work Management subsystem then changes the status of any follower tasks that directly depend on the task just completed from Pending to Ready.

Work Management Operational Differences

Table 14–8 lists the similarities and differences between the operation of the Work Management API and the Work Management subsystem.

Table 14–8 Work Management Subsystem and Work Management API Differences

Key Work Management Function	WM Subsystem	WM API
Define provisioning plans	Yes	No
Define rules and behaviors by which MetaSolv Solution can dynamically change a provisioning plan	Yes	No
Apply previously defined provisioning plans to service requests	Yes	Yes
Add tasks to, copy tasks within, or remove tasks from a service request after the provisioning plan has been applied	Yes	No
Mark a task **Required** or **Not Required**	Yes	No
Define the dependency relationships among the tasks on a provisioning plan	Yes	No
Accept a task	Yes	Yes
Add, remove, or modify the dependency relationships between the tasks assigned to a service request	Yes	No
Define the default due dates and duration for tasks	Yes	No

Table 14–8 (Cont.) Work Management Subsystem and Work Management API

Key Work Management Function	WM Subsystem	WM API
Modify the due date for tasks at the time of task generation	Yes	No
Change the estimated completion date for a task	Yes	Yes
Display task due times that are adjusted for the user's local time	Yes	Yes
Assign tasks to the default work queues defined in the selected provisioning plan	Yes	Yes
Override the default work queue assignments for one or more tasks at the time of task generation	Yes	Yes
Dispatch tasks into work queues	Yes	Yes
Track information for one or more of the tasks assigned to a specified service request, including each task's current status, due date, dependency relationships, and the work queue to which it has been assigned	Yes	Yes
Track the tasks in a specified work queue	Yes	Yes
Transfer tasks from one work queue to another	Yes	Yes
Set a task's status to Complete using the current date/time	Yes	Yes
Set a task's status to Complete using an earlier date/time	Yes	Yes
Auto-complete tasks marked as auto-completable when their follower task is completed	Yes	Yes
Auto-complete tasks marked as auto-completable at the time of task generation when the task has no follower task	Yes	No
Re-open a completed task	Yes	Yes
Reject a task that was completed earlier and return the rejected task to the work queue of the employee or work group that initially completed the task, along with the reason for the rejection	Yes	Yes
View task detail information for a given task	Yes	Yes
Assign jeopardy codes to tasks or to circuits associated with a task	Yes	Yes
Assign notes to an order or to circuits on a task	Yes	Yes
Report why a task was completed late	Yes	Yes
View the service request detail	Yes	Yes
View notes that have been added to the service request	Yes	Yes

Tasks That Cannot be Completed Through the Work Management API

The Work Management API cannot complete any tasks that are in Pending status.

When the tasks from the following list are in Ready status, they cannot be completed through the Work Management API even though they can be completed through the Work Management subsystem:

- CAD: Carrier Access Billing System (CABS) Acknowledgment Date task
- CID: CABS Issue Date task
- DD: Due Date task

> **Note:** When configured to do so, the MetaSolv Solution System Task server can automatically complete DD tasks. However, the Work Management API does not handle the completion of the DD task. If you attempt to complete a DD task through the Work Management API, the API returns an exception.

- EUAD: End User Billing Acknowledgement Date task
- EUID: End User Billing Issue Date task

All other MetaSolv Solution-defined tasks in Ready status and all customer-defined tasks in Ready status can be completed through the Work Management API.

Work Management API Support for NET DSGN Task

The Work Management API supports the NET DSGN (Network Design) task type. API users are able to accept, complete, transfer, and reject NET DSGN tasks when the tasks are included in a provisioning plan just as if they were using the MetaSolv Solution GUI.

Work Management API Support for Date Ready System Tasks

The Work Management API supports the date ready system task feature in the MetaSolv Solution Work Management subsystem.

When API users use the Work Management API to generate and assign a task that the provisioning plan identifies as a date ready system task, the System Task Server does not process that task until its scheduled start date, just as if the task had been generated and assigned using the MetaSolv Solution windows.

Work Management API Support for Backdated and Forward-dated Tasks

The Work Management API supports the backdated and forward-dated task features in the MetaSolv Solution Work Management subsystem.

When an access service request (ASR) or local service request (LSR) is created, the due date (DD) task must be scheduled before the order recipient confirms the order. Backdating and forward-dating means that the tasks after the CONF or RCONF task are rolled forward or backward to accommodate the confirmed dates. See the online Help for more information.

A

API Error Messages and Exceptions

The APIs return error messages by raising exceptions that use the WDIExcp or WDIError structures. The error number identifies the general area in which the error occurred.

Table A–1 lists the error number ranges and their related areas.

Table A–1 Error Number Ranges and Related Areas

Error Range	Impacted Area
10000 to 19999	System errors that can occur in all APIs.
20000 to 29999	LSR API
30000 to 39999	PSR API
40000 to 49999	PSR Ancillary API
60000 to 69999	Internet Services APIs
70000 to 79999	Trouble Management API
80000 to 89999	Work Management API
90000 to 99999	Miscellaneous errors. These are not necessarily system related but could be encountered by multiple applications.

The **reason** field in the WDIError and WDIExcp structures provides a text description of the error condition.

B

Tips And Techniques

This section describes tips and techniques that you can implement to effectively use the CORBA APIs.

Understanding IOR Files

The APIs can be configured to use IOR files to route events to external applications. Your API System Administrator can tell you whether the APIs use IOR files in your environment.

IOR files are either created or overwritten if they already exist when the Oracle Communications MetaSolv Solution Application Server is brought up. IOR files contain a Stringified object reference that encodes the IP address of the server that hosts the object and additional information specific to the ORB vendor that identifies the object reference.

Configuring the IOR File to Enable External Systems to Connect to the CORBA Server

When connecting to the CORBA server from external systems, you may receive the following error message:

```
ERROR: Unable to Connect to Manager: org.omg.CORBA.TRANSIENT: Retries exceeded,
couldn't reconnect to 10.196.7.146:2512  vmcid: 0x0  minor code: 0   completed: No
ERROR: org.omg.CORBA.TRANSIENT: Retries exceeded, couldn't reconnect to <target ip
address>:<target port>  vmcid: 0x0  minor code: 0  completed: No
```

You may observe this issue if the CORBA server is hosted on a computer that has enabled two IP addresses; an internal IP address and an external IP address. If you install the MSS application on such a computer and enable the CORBA server, the internal IP address of the computer is copied in the IOR file when you create it. However, the internal IP address is not detected by the external systems trying to connect to the CORBA server on any given network. As a result, the external systems fail to successfully connect to the CORBA server and you receive the error message.

The internal and external IP addresses of a computer may differ, but the host name of the computer always remains constant. Therefore, for external systems to successfully connect to the CORBA server, you must specify the host name of the computer (instead of its IP address) in the IOR file.

To specify the host name of the computer in the IOR file, you must enable the jacorb.dns.enable property in both the **orb.properties** and **jacorb.properties** files, as follows.

```
jacorb.dns.enable=on
```

The default value of the jacorb.dns.enable property is off.

CORBA.INV_OBJREF and CORBA.OBJECT_NOT_EXIST Exceptions

A CORBA client can receive CORBA.INV_OBJREF or CORBA.OBJECT_NOT_EXIST exceptions when it attempts to use a Stringified IOR object reference or any other remote object reference in any of the following cases:

- The CORBA server that hosts the object has been brought down
- The CORBA client has already called a destructor method in the IDL that destroys that object
- The CORBA server times out

A typical situation where this might occur is where the server is re-configured to write IOR files to a new location, but the client is still reading old IOR files from the original location.

CORBA.COMM_FAILURE Exception

The most common cause of a CORBA.COMM_FAILURE exception is the remote ORB daemon being down, in addition to a server termination during a client operation. A newly-obtained remote reference is generally not validated during the string_to_object or narrow() operations. Instead, for the sake of efficiency, creation of a physical connection to the remote host is delayed until the first IDL operation is actually invoked. This is the point at which most remote exceptions occur. For Java ORBs, NullPointerExceptions during these initial connection operations can be the result of invalid mixed stub class and ORB class types or versions, as well as an attempt to marshal a structure (that is, to pass a parameter to a remote object) that contains a null string or other null element.

Using the MetaSolv Solution APIs With Multi-Threaded Clients

To use the API in a synchronous application such as a Web page, a multi-threaded client may use a mechanism (such as Java's *Object.wait*) in the calling method and *Object.notify* or *Object.notifyAll* in the notification object's result methods, which can be called by the ORB in a separate thread. When the result is returned, the calling thread wakes up, gets the data from the notification object, and continues processing. The notification object itself can be used as the monitor object.

Developing Using C++

When your application is developed in C++, the general principles remain the same. You must develop code to implement the callback object, and when linking object files to create an executable, must link in both the client and server libraries provided by the ORB vendor.Synchronization Primitives and C++.

The C++ language does not provide synchronization primitives. If you use C++, you must use primitives supplied by the host operating system (for example, semaphores) to achieve the desired result.

C++ Troubleshooting

If you are using C++ as your implementation language:

- You must use a compiler version that supports the namespace feature.

- You may encounter a problem while compiling the MetaSolv Solution IDL-generated source in C++ where CORBA primitive types are not found.

  This can be traced to a problem with the IDL generator. C++ namespace resolution assumes that a CORBA::type, such as CORBA::char will be defined within the MetaSolv::CORBA namespace. The MetaSolv Solution API naming conventions and Java-based development do not permit this.

  If you encounter this problem you must completely scope CORBA::types. For example, if the generated code is:

  ```
  typedef CORBA::char null;
  ```

  You should change the code to:

  ```
  typedef ::CORBA::char null;
  ```

Troubleshooting Tips for API Developers

1. Review the documentation.
2. Confirm that you have used appropriate development techniques:
3. Design application and exception logging and stack tracing into your application. Make sure you have exception logging for all these cases:
   - CORBA exceptions
   - API exceptions
   - Logic exceptions
   - Control logging using .ini or command line parameters.
4. Dump structure traces before and after export or import.
5. Use existing logging capabilities:
   - API server logging
   - SQL logging
   - Console logging
   - CORBA logging

Using API Server Logging

There is only one log file (**jacORB.mss.log**) for all the CORBA API servers.

1. Check the appropriate server's error log.
2. If required, set the LoggingOn parameter to **true** in the **gateway.ini** file.
3. If required, set the TraceLevel parameter in the **gateway.ini** file to the required value:
   - 0 = High level logging. Only error information.
   - 1 = More detailed logging includes error information and some system information.
   - 2 = Most detailed logging. Includes much more system information.

> **Note:** Only use high trace levels when debugging, because these options generate significant amounts of information.

4. Log returned data.
5. If required, set the PrintExportObjects parameter to **true** in the [Gateway] section in the **gateway.ini** file. This logs the values in objects in IDL structure format.

> **Note:** This option generates significant amounts of information.

Using SQL Logging

1. Log SQL statements.
2. If required, set the SQLLogging parameter to **true** in the [System] section in the **gateway.ini** file.
3. Verify database related issues.

> **Note:** This option generates significant amounts of information.

Using Console Logging

For system errors, use Oracle Administration Console logging. This option reports extreme exception cases. For more information, see the documentation on logging.

Using CORBA Logging

1. Use CORBA diagnostics provided by the ORB.
2. Know your ORB Tracing mechanisms.
3. Check CORBA diagnostics:
4. Determine the source of connections.
5. Determine the methods invoked.
6. Diagnose connection information.

For more information, see ORB documentation.

C

Sample Code

This appendix provides sample code to help you gain a better understand of using the Oracle Communications MetaSolv Solution APIs.

IOR Bind Method

This section provides information about the IOR bind method.

Background

The API architecture and the MetaSolv Solution Application Server architecture both support IOR binding. This type of binding is supported by the server processes creating a flat file containing the stringified object reference. A client can read this object reference from the file, convert it to a real object reference using a CORBA function, and connect to the server.

To create IOR files, the INI parameters are changed in the **gateway.ini** file. These parameters are the StrictOMG (API only) and IORPath settings. In the API architecture, the StrictOMG parameter needs to be set to true. In the Application Server platform, this parameter is ignored and the IOR is always produced. The IOR is written to the location specified in the IORPath statement. These files are named using the server name with an extension of IOR. These files are produced when the server is initialized.

The server writes the IOR file to disk. Client machines that want to use this mechanism must access this file. There are two ways to accomplish this; distribute the IOR file to the client machine or have the client and server access a shared-drive location. Distributing the IOR can be very problematic because the IOR file is recreated every time the server is restarted. The distribution process must account for this condition. The shared-drive method avoids this problem because both the client and the server access the same drive location. Oracle recommends using the shared-drive approach to access the IOR file.

IOR Bind Method Sample Code

This code example illustrates how to use the IOR binding mechanism.The “Hello API” sample application code provided with the MetaSolv Solution APIs uses the IOR binding mechanism. The following steps are required to set up the IOR bind mechanism:

1. First set gateway.ini parameters in order for MetaSolv Solution to create an IOR file:

```
[System]
IORPath=<directoryname>
```

```
//Locate the bind code. The code should look something like this.
ORB.init(args, null);
String hostname = "MetaSolv Solutionapihost"; //machine name of API host
String servername = "DLRSERVER"; //MetaSolv Solution API CORBA
//server name
try {
WDIRoot aWDIRoot = WDIRootHelper.bind(":"+servername, hostname);
}
catch (SystemException se) {
System.out.println("Unable to bind to server: " + se);
)
MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = aWDIRoot.connect(req);
```

2. Change the code found in item two to read the IOR from a file, convert it to an object, and narrow the scope of the object. After the object is narrowed, then processing can continue as usual. This logic is not placed in program order. For the exact code, reference the sample applications.

```
orb = ORB.init(args, null);
// Connect to the DLR API Server and construct a proxy for the
// root object.
String iorfile = System.getProperties().getProperty(DLR_IOR_FILE_PROPERTY);
// Set a system property on command line using -D (for Sun) or /d: (for MS)
// Block A ////////////////////////////////
if (iorfile == null)
throw new Exception("'" + DLR_IOR_FILE_PROPERTY + "' system
property not set on command line.");
System.out.println("IOR file="+iorfile);
String ior = readIOR(iorfile);
System.out.println("DLR IOR="+ior);
//////////////////////////////////////////////
```

The block of code above marked Block A shows the changes required. Through standard Java file operations, the IOR string produced by the server is inserted. The name of the file is passed in from a command line parameter. Remember, the client must have access to the IOR file produced by the server. The best way to do this is by accessing a shared drive. The data is read using the readIOR function below.

Here is an extract of the readIOR function. This contains standard Java code to read a file:

```
private static String readIOR(String fileName) throws IOException
    {
 byte[] iorBytes = new byte[5000];
int size = 0;
FileInputStream fs = new FileInputStream(fileName);
      try {
size = fs.read(iorBytes);
 } finally {
fs.close();
      }
 return new String(iorBytes, 0, size);}

//Block B////////////////////////////////
org.omg.CORBA.Object obj = orb.string_to_object(ior);
WDIRoot aWDIRoot = (WDIRoot)WDIRootHelper.narrow(obj);
```

```
//////////////////////////////////////
```

The block of code marked Block B above converts the string read from the file into an object reference. This is done using the string_to_object reference. The narrow function takes that object reference and casts it to the correct object type. Once this is done, the remaining code is the same. The code captured below shows how the object reference is used to access other methods. This is the same code used in the bind method without any additional changes.

```
MetaSolv.CORBA.WDI.ConnectReq req = new
MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = aWDIRoot.connect(req);
```

NameService Bind Method

This section provides information about the NameService bind method.

Background

A CORBA NameService is a mechanism defined in the CORBA standard for connecting clients and servers. The NameService provides an "index" of available servers to which it can connect. Each of these servers are found through the use of a name.

The MetaSolv Solution Application Server enables a NameService to facilitate this binding method. There are three methods used to locate the NameService: the IOR file method, the ResolveInitialContext method, and the URL method.

The IOR method of finding the NameService is similar to the process described in the previous section, a file is read that contains the IOR. The IOR is then converted to an object reference for use by the NameService. Sample code is shown later in this section for this bind mechanism.

The resolve_initial_references method of finding the NameService is also available. The resolve_initial_references method is an OMG standard for identifying the NameService. The configuration parameters used by various CORBA vendor's software to find the NameServer varies. Review the documentation provided by your CORBA vendor for details on how the NameService is found. Sample code is shown later in this section for this bind mechanism.

The third method a third party can use to bind to an API running in the MetaSolv Solution Application Server is the URL Bind Method. With this method, an HTTP can be used to get the stringified IOR from the NameService. This type of binding is only supported in the new Application Server architecture.

1. Your first step is to locate the bind code. The code should look something like this:

```
ORB.init(args, null);
String hostname = "MetaSolv Solutionapihost"; // machine name of MetaSolv
//Solution API host
String servername = "DLRSERVER"; // MetaSolv Solution API
//CORBA server name
try {
//Block C//////////////////////////////////////
WDIRoot aWDIRoot = WDIRootHelper.bind(":"+servername, hostname);
//////////////////////////////////////////////
}
```

```
catch (SystemException se) {
System.out.println("Unable to bind to server: " + se);
)
MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = aWDIRoot.connect(req);
```

2. The next step is to replace it with one of the following two methods:

Binding to the NameServer With an IOR Sample Code

```
//Block D ////////////////////////////////////
org.omg.CosNaming.NameComponent[] name;
////////////////////////////////////////////
// Connect to the NameService using the IOR. Published by the
// MetaSolv Solution Application Server
// get the command line parameter
String iorfile = System.getProperties().getProperty(NS_IOR_FILE_PROPERTY);
// Set a system property on command line using -D (for Sun) or /d: (for MS)
if (iorfile == null)
throw new Exception
("'" + NS_IOR_FILE_PROPERTY + "' system property not set on command line.");
```

The previous block of code defines the name component variable. The first line of the sample code is an OMG standard variable used for the lookup in the NameService. This variable is used later. The rest of this code obtains the location of the NameService IOR file. The location is passed in as a parameter on the command line of the program. The NameService IOR is written to the location specified in the IORPath statement of the **gateway.ini** file. This NameService IOR file is named NAMESERVICE.IOR. The best way to locate the file is to have the client and server access a shared-directory location.

```
System.out.println("IOR file="+iorfile);
String ior = readIOR(iorfile); //read the IOR
System.out.println("NS IOR="+ior);
org.omg.CORBA.Object obj = orb.string_to_object(ior); //convert to object ref
org.omg.CosNaming.NamingContext rootContext =
org.omg.CosNaming.NamingContextHelper.narrow(obj); //narrow the object
System.out.println("loaded orb class:"+orb.getClass().getName());
```

The previous block of code reads the IOR contained in the file. It converts the IOR to an object reference and narrows the object reference to a CosNaming object. After these steps are completed, the program has the reference to the NameService that is running on the Application Server. The program can now use the NameService to look up the CORBA server it is using.

```
//populate the name component for lookup
(block A)

name = new org.omg.CosNaming.NameComponent[1];
name[0] = new org.omg.CosNaming.NameComponent("DLRSERVER", "");

// "lookup DLR Server";

(block B)

org.omg.CORBA.Object dlrobj = rootContext.resolve(name);
// narrow the object ref for dlr server
```

```
WDIRoot aWDIRoot = (WDIRoot)WDIRootHelper.narrow(dlrobj);
```

The next step in the process is to look for the actual CORBA server you want to bind to and obtain its object reference from the NameServer. This process is shown in the previous block of code (code block A).

Create the objects used to query the NameService. The Application Server uses a single level of naming. The **gateway.ini** file controls these names. The "servers" section connects the name in the NameService to the object. For example:

```
DLRSERVER=MetaSolv.CORBA.WDIDLR.WDIRoot,MetaSolv.WDIDLR.WDIRootImpl
```

The name "DLRSERVER" is registered in the NameService for the object on the right side of the "equals" sign. In the previous code fragment, "DLRSERVER" was used as the name.

The resolve method is used to look up the object in the NameService (code block B). The resolve method returns a standard CORBA object. This object then needs to be further refined using the CORBA narrow method. After using this command, the root object reference is obtained and the other methods on the server are used. The code fragment below shows this functionality:

```
MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
System.out.println("Connecting to MetaSolv Solution API Server...");
WDIManager aWDIManager = aWDIRoot.connect(req);
```

Binding to the NameService with resolve_initial_references Sample Code

This method of finding the NameService relies on the CORBA standard of resolve_ initial_references. This method is available once the ORB has been initialized and provides the object reference of the NameService. Although this method is a CORBA standard, each CORBA vendor implements a different way to locate the NameService. Most of the time configuration parameters such as .INI files are used. Refer to your CORBA vendor's documentation for more details. The following code sample shows how to use this method to connect to the NameService. This code assumes the configuration parameters are set to identify the location of the NameService:

```
org.omg.CosNaming.NameComponent[] name;

org.omg.CORBA.Object obj = orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext rootContext =
org.omg.CosNaming.NamingContextHelper.narrow(obj); //narrow the object
System.out.println("loaded orb class:"+orb.getClass().getName());
```

The previous block of code defines the name component variable. This variable is an OMG standard variable used to do the look up in the NameService. It is used later. Next, the object reference of the NameService is obtained and stored in a CORBA object. This is done using the resolve_initial_references method passing in the service name. The CORBA standard for the NameService service is the string "NameService." To be useful, cast it into a CosNaming object. This is accomplished using the narrow method. The program can now use the NameService to look up the CORBA server it is using.

```
//populate the name component for lookup
name = new org.omg.CosNaming.NameComponent[1];
name[0] = new org.omg.CosNaming.NameComponent("DLRSERVER", "");
// "lookup DLR Server";
```

```
org.omg.CORBA.Object dlrobj = rootContext.resolve(name);
// narrow the object ref for dlr server
WDIRoot aWDIRoot = (WDIRoot)WDIRootHelper.narrow(dlrobj);
```

The next step in the process is to look for the actual CORBA server you want to bind to and obtain its object reference from the nameserver. This process is shown in the previous block of sample code. Now create the objects used to query the NameService. The Application Server uses a single level of naming. The **gateway.ini** file controls these names. The "servers" section connects the name in the NameService to the object. For example:

```
DLRSERVER=MetaSolv.CORBA.WDIDLR.WDIRoot,MetaSolv.WDIDLR.WDIRootImpl
```

The name "DLRSERVER" is registered in the NameService for the object on the right side of the "equals" sign. In the previous code fragment, "DLRSERVER" was used as the name.

To look up the object in the NameService, the resolve_initial_references method is used. The resolve method returns a standard CORBA object. This object is refined further using the CORBA narrow method. After using this command, the root object reference is obtained and the other methods on the server are used. The code fragment below shows this process:

```
MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
System.out.println("Connecting to MetaSolv Solution API Server...");
WDIManager aWDIManager = aWDIRoot.connect(req);
```

URL Bind Method Sample Code

The third method used to bind to an API running in the MetaSolv Solution Application Server is the URL Bind Method. With this method an HTTP can be used to get the stringified IOR from the NameService. This type of binding is only supported in the new Application Server architecture.

To use this method of obtaining the IOR, an .INI parameter is enabled in the **gateway.ini** file. This parameter is the URLNamingServicePort settings. This parameter is located in the system section of the **gateway.ini** file. By default this parameter is commented. To uncomment it, remove the semicolon and restart the machine.

Once the parameter is enabled, a standard URL request can be used to return the IOR reference. The reference returned is to the API server. So you must format the URL request in the following way:

```
http://hostname:15000/DLRSERVER
```

- *hostname*: The hostname contains either the IP address or host name of the machine running the API architecture.
- 15000: The port defined in the **gateway.ini** file for the URLNamingServicePort parameter. By default the value is 15000.
- *DLRSERVER*: The name of the desired server object. This is defined in the **gateway.ini** file in the servers section.

When this request is complete the IOR is returned. This method can be tested using any browser by typing the URL address in the location field. The browser returns the IOR. After the IOR is returned the same code used in the IOR bind method section is

used. The only difference in the procedure is the file access to read the IOR can be omitted.

Sample Code

1. Set gateway.ini parameters to activate this processing in the MetaSolv Solution Application Server. This activation is done by uncommenting the line in the .INI file:

```
[System]
URLNamingServicePort=15000
```

2. Locate the bind code. This bind code should look something like this.

```
ORB.init(args, null);
String hostname = "MetaSolv Solutionapihost"; // machine name of MetaSolv
//Solution API host
String servername = "DLRSERVER"; // MetaSolv Solution API
//CORBA server name
try {
WDIRoot aWDIRoot = WDIRootHelper.bind(":"+servername, hostname);
}
catch (SystemException se) {
System.out.println("Unable to bind to server: " + se);
)
MetaSolv.CORBA.WDI.ConnectReq req = new MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = aWDIRoot.connect(req);
```

3. Change the code found in item two to read the IOR from the naming service URL, convert it to an object, and narrow the scope of the object. After the object is narrowed the processing can continue as usual.

```
orb = ORB.init(args, null);

// Connect to the DLR API Server and construct a proxy for
// root object.
String URLref = System.getProperties().getProperty(URL_IOR_FILE_PROPERTY);
// Set a system property on command line using -D (for Sun) or /d: (for MS)
//Block E ////////////////////////////////////////////
URL url = new URL(URLref);
URLConnection conn = url.openConnection();
int len = conn.getContentLength();
if (len == -1) throw new IOException("URL not found: ["+sUrl+"]");
InputStream in = (InputStream)conn.getContent(); // get the data from the
request
byte[] bior = new byte[len];
in.read(bior);// read IOR from input stream
String ior = new String(bior);// convert to string
//////////////////////////////////////////////////////
```

The block of code marked Block E above shows the required changes. The URL to connect to is passed in from a command line parameter.

```
org.omg.CORBA.Object obj = orb.string_to_object(ior);
WDIRoot aWDIRoot = (WDIRoot)WDIRootHelper.narrow(obj);
```

The block of code above converts the string retrieved from the URL into an object reference. This is done using the string_to_object reference. The narrow function takes

that object reference and casts it to the correct object type. Once this is done the remaining code is the same. The code captured below shows how the object reference is used to access other methods. This is the same code used in the bind method:

```
MetaSolv.CORBA.WDI.ConnectReq req = new
MetaSolv.CORBA.WDI.ConnectReq();
//The following values are only examples of the user name and password values.
req.userName = "ASAP";
req.passWord = "ASAP";
WDIManager aWDIManager = aWDIRoot.connect(req);
```

Gateway Events Functionality Changes

The MetaSolv Solution gateway event functionality is also changing for this release. The changes are being made to allow for better integration with the new Application Server and to provide additional functionality. Unlike the changes described in previous sections, these change impact both the API architecture and the MetaSolv Solution Application Server architecture.

Gateway events provide a mechanism for MetaSolv Solution to notify a third-party application to start an activity. The third-party application receives this event through a CORBA method call. In this case, MetaSolv Solution is the client and the third-party application is the server. After receiving the call, the third-party application performs the appropriate actions. There are three types of gateway events in previous releases. The gateway events are:

- Standard Gateway Event: The standard gateway event is an event defined by a third party. When the event is invoked, the third party is notified the event has occurred and key related data is passed to the third party. These events can be related to both work plans, PSRs, and trouble tickets.
- Billing Gateway Events: The two billing gateway events send billing information to the third party. These two events are Send Billing Customer (SBC) and Send Billing Order (SBO). These events differ from the standard gateway event because they contain billing data and are not defined by the third party.
- SOA Gateway Events: The SOA application contains many gateway events and is used to interact with a third-party LSOA application. These events differ from a standard gateway event due to the type of data they send and the fact they are tightly coupled to MetaSolv Solution. These events are not defined by the third party.

For more detailed information on gateway events see "Signal Handling Pattern".

Middle-tier Triggering

MetaSolv Solution provides the ability to trigger standard gateway events from the middle-tier server. This behavior occurs in both the API and Application Server environments, which provides a robust triggering method that has many advantages, including maintenance of fewer connections to the third party, enabling automatic retries, and requiring minimum client configuration.

To enable this functionality, two new statuses were added to the gateway events. These statuses indicate the intermediate state between the user asking for a triggering event and the event actually being sent to the third-party applications. A user sees these states in the gateway event status windows located in the work management queue, the PSR order window, and the Trouble report window. These states are:

- Sending: An event in Pending status has been triggered by the user or auto-signaled. The status of the event has changed, but not picked up by the middle-tier triggering server and communicated to the third-party application. When the middle-tier triggering server receives the event and communicates it to a third-party application, the event status changes to "Waiting". If the event cannot be communicated to the third-party application, the status of the event changes to Error.
- Terminating: An event in Waiting or In-progress status has been “reactivated” by the user using the GUI. These two statuses indicate the third-party application is processing the event. The fact the user has terminated the event must be communicated to the third-party application. The terminating state is the transition state before the event is returned to Pending status and indicates the third-party application was not notified of the status canceling. Once the middle-tier server can communicate this status, the event is changed to Pending.

No third-party coding changes are required to support this new processing. The middle-tier server sending the events uses the same CORBA methods used in the past. The only impact you see is a reduced number of connections. However, some additional parameters are captured when a gateway is defined. This process is described in the next section.

This change only applies to the standard gateway events. The billing and SOA events are not yet converted to this architecture. These events are still invoked directly from the client to the third-party application. This transition will be made in a later release.

New Binding Methods

This section provides information about the new binding methods.

Background

As with the MetaSolv Solution Application Server, the MetaSolv Solution gateway event architecture is moving to an OMG standard for binding the client to the server.The reason for this change is to enable the choice of open technologies for Oracle and Oracle customers. IOR binding, and NameService binding are supported.

To enable this support the method of defining gateways in MetaSolv Solution has changed. Additional parameters are defined. Changes are required in the third-party applications to allow the CORBA standard binding. These changes are standard CORBA coding techniques and detailed later in this section.

Defining a Gateway

The way you define a gateway in MSS enables you to trigger events from the middle-tier server and define multiple bind locations.

Figure C–1 shows the Gateway Events window.

Figure C–1 Gateway Events Window

The Gateway Events window contains the following fields:

- **Number of Retries**: When a standard gateway event is in Sending status, the middle-tier event server attempts to trigger the event to the third-party application. If the event cannot trigger, a retry process is started. This field indicates how many times a triggering event is retried before the event is marked in error. The default is five.
- **Retry Interval (secs)**: This field indicates the delay before trying to resend an event. This time interval is designed to enable the third-party application time to recover. The default is 30 seconds.
- **User ID**: This field is required to connect to the third-party application. This field allows one user ID for all connections to the third-party application. This **User ID** field was added because the middle-tier event server does not have the user ID and password of the user who initiated the event or the definition of every MetaSolv Solution user in the third-party application. The only impact to third-party applications is if they used this user ID for identification of the person initiating the event, this user ID needs to change to the user ID contained within the WDIEvent occurred structure sent with the event (see below). This user ID contains the identification of the actual user who initiated the event. See the code sample below:

```
struct WDIEvent
    {
    long    eventVersion;
    string  eventName;
    long    documentNumber;  /// An oracle generated sequence that uniquely
                             /// identifies a Service Request in the MetaSolv
                             /// Solution database.
    long    taskNumber;      /// An oracle generated sequence that uniquely
                             /// identifies a task in the MetaSolv Solution
                             /// database.
```

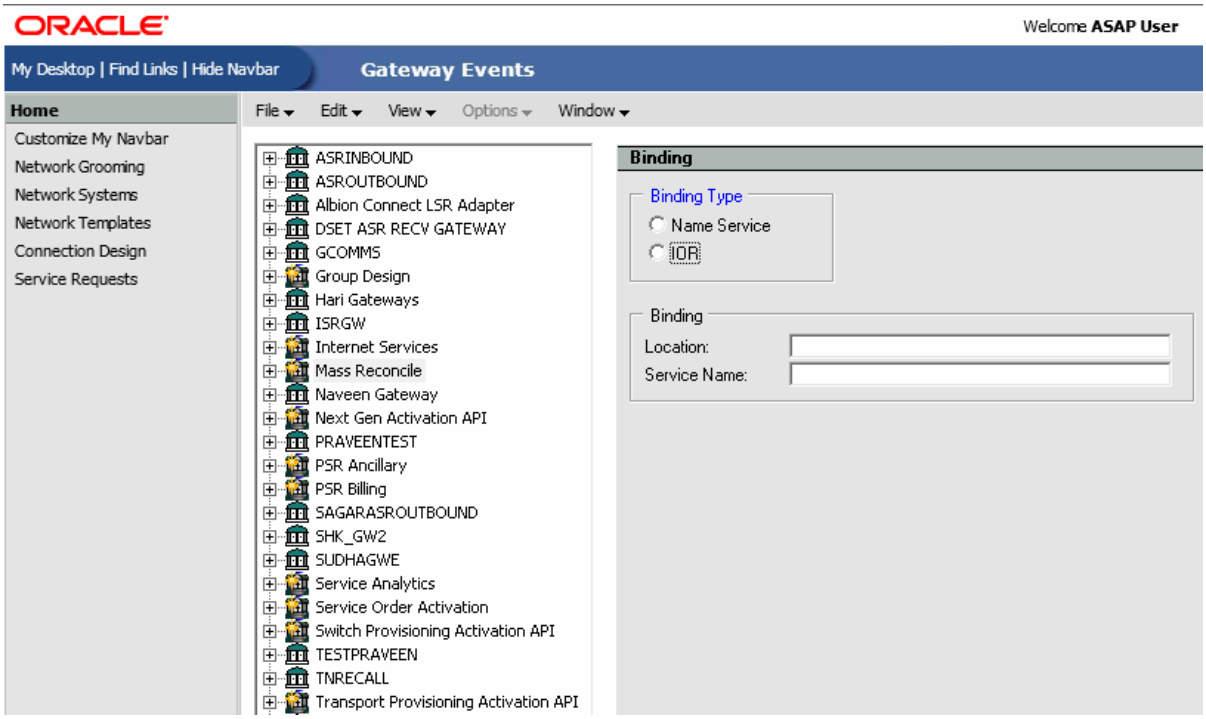
```
long    servItemID;      /// An oracle generated sequence that uniquely
                         /// identifies a service item in the MetaSolv
                         /// Solution database.
                         /// Note this value is only supplied for an
                         /// item level gateway event.
string  userID;          /// A MetaSolv Solution user ID.
};
```

During the upgrade new data is created and the values are used as defaults. If the defaults need to change, each gateway can be edited using the Gateway Events window. The defaults are:

- Number of Retries: 5
- Retry Interval (secs): 30
- UserID: Default

The second change made to gateway maintenance is to enable the definition of multiple binding locations and type of binding to use for each gateway binding location. The type of binding allows the identification of the connection to that gateway. The options are IOR and NameService. Another change is functionality that allows gateways to have multiple bind locations defined. This enables the MetaSolv Solution event architecture to try different locations if it cannot bind to a server. Figure C–2 illustrates binding options:

Figure C–2 MetaSolv Solution Gateway Binding Window



The treeview on the left now has the binding information defined. The multi-color lightning bolt represents a binding option. For each binding option, detail information is captured. This is shown on the right side of the window. The following list details the information:

- Binding Type: Indicates what binding mechanism to use for this binding location. The options are IOR and NameService.
- Binding Location: Indicates the path to locate the IOR for binding. This path is used for IOR and NameService binding. This can be expressed as a standard windows path (C:\ior\files) or a URL (http://srvg/ns.ior).
- Binding Service Name: This field indicates the name of the service to request to obtain the WDIRoot reference if using NameService binding.

When multiple binding locations are defined, the bind process attempts to connect them in the order defined in the tree view on the left. Each bind location can use a different binding approach.

During the upgrade, a binding location is created. This binding location has the same host and server name used for previous release. Only one binding location is created.

WARNING: There is no immediate impact to third-party applications by these changes. However, when a customer migrates from the API architecture to the MetaSolv Solution Application Server, the binding process should be changed to CORBA standard binding. The third-party application must transition to either IOR binding or NameService binding as detailed below.

IOR Binding to Third-party Applications

IOR binding requires the third-party application to produce an IOR read as a text file. The gateway event application locates that file and connects to the server. The IOR produced needs to represent the WDIRoot object of the server.

Ensure IOR published is the IOR of the WDIRoot object. This object reference is available after the object is connected to the ORB. This object is then converted to a string and written to a file. The following code shows this and is taken from the hello_ gateway server sample code shipped with the documentation. This code fragment is often placed in the main class:

```
orb = ORB.init(args, null);
WDIRoot aWDIRoot = new WDIGatewayRootImpl();
 orb.connect(aWDIRoot);  // Some ORBs (e.g. JacORB) require an explicit connect
```

The previous block of code creates the server and registers it to the ORB. The variable string iorfile = System.getProperties().getProperty(GATEWAY_IOR_FILE_ PROPERTY);

```
// Set a system property on command line using -D (for Sun) or /d: (for MS)
 if (iorfile == null) {
       System.out.println("'" + GATEWAY_IOR_FILE_PROPERTY + "' system property not
set on command line.");
               return;
            }
```

The previous block of code determines where to write the IOR file. For this sample program, the location is passed to the program by a command line parameter. This location must be accessible by both the middle-tier event server and the server that produces it. These requirements are detailed in the following section.

```
writeIOR(orb.object_to_string(aWDIRoot), iorfile);
```

The previous block of code calls a function to create the file. The object_to_string function converts the object reference to a string. This string can then be written to a file. The client that calls this server can then use this IOR to connect to the server. For details on how this works, see the API Code Transition section of this document or your CORBA documentation.

NameService Binding to Third-party Applications

NameService binding provides another method of locating the server. To use this type of binding the third-party application must support a NameService. To accomplish this, the NameService process must run in the ORB and the third-party application must have registered its process in the NameService. For details on how to accomplish this, refer to programming documentation provided by your CORBA vendor.

At this time, the gateway event architecture does not support the resolve_initial_ references process of finding the NameService of the third-party application. An IOR of the NameService is required. As a result, the third-party application must capture the IOR of the NameService and write this IOR to the file system. The following code fragment shows how to capture the IOR of the NameService. This code must be run immediately when a third-party environment is activated. It is not required for every server since the NameService is global. Typically, this is done during a global start-up process.

```
org.omg.CORBA.Object obj = orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext rootContext =
org.omg.CosNaming.NamingContextHelper.narrow(obj); //narrow the object
```

The previous block of code connects to the NameService. The object reference of the NameService is captured in the obj variable.

```
String iorfile = System.getProperties().getProperty(GATEWAY_IOR_FILE_PROPERTY);
// Set a system property on command line using -D (for Sun) or /d: (for MS)
 if (iorfile == null) {
       System.out.println("'" + GATEWAY_IOR_FILE_PROPERTY + "' system property not
set on command line.");
                return;
            }
```

The previous block of code determines where to write the IOR file. For this sample program, the location is passed to the program by a command line parameter. This location must be accessible by both the middle-tier event server and the server that produces it. These requirements are detailed in this section.

```
writeIOR(orb.object_to_string(obj), iorfile);
```

The previous block of code calls a function to create the file. The object_to_string function converts the object reference to a string. This string is then written to a file. The client that calls this server can then use this IOR to connect to the NameService. For more detailed information, see the API Code Transition section of this document or your CORBA documentation.

New Event Signal

MSS supports a new event signal. This server is used to support trouble events, including other events. All event processing from previous releases continue to use the same IDL methods used in previous releases. There should be no transition steps to maintain existing functionality.

D

The PSR End User Billing API

The PSR End User Billing (PSREUB) API provides a mechanism for exporting data from the Oracle Communications MetaSolv Solution database to support end-user billing from product service requests (PSRs). This API defines a standard end-user billing interface which enables you to develop applications that act as a mediation layer to third-party billing systems. MetaSolv Solution facilitates integration with third-party billing applications by notifying the billing application when:

- Gateway events associated with end user billing are initiated
- An existing customer account is modified

Implementing an third-party billing server requires a detailed understanding of the PSREUB API as well the data communication and translation protocols involved in communicating with the billing system. The scope of this documentation is confined to a discussion of how to implement the interfaces required to integrate with the PSREUB API.

Note: Before the PSREUB API can export any account or order information, a considerable amount of information must be set up correctly in the MetaSolv Solution database.

Essential Terminology

Table D–1 defines the terms that identify the concepts and information that are required to understand the PSREUB API.

Table D–1 MetaSolv Solution Concept Terms and Information

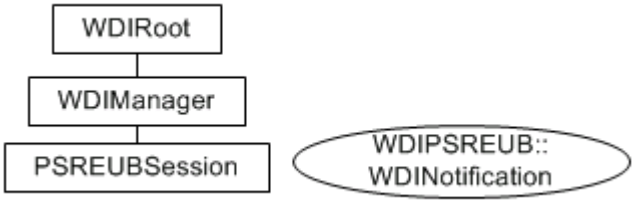
Term	Definition
Billing interface code	A status code housed in the MetaSolv Solution database but maintained by the third-party billing server. This code indicates whether the customer account information has been communicated to the billing system. Actual values used can vary depending on the billing system used.
System-defined gateway event	These are gateway events used by the PSREUB API. These events are pre-defined in MetaSolv Solution. Users can view these events on the MetaSolv Solution Gateway Maintenance window, but cannot modify event attributes.
third-party billing server	An application developed by you or a third party that allows MetaSolv Solution to communicate end user billing data to a third-party billing system.

PSREUBSession Interface

The PSREUBSession interface is defined in the WDIManager interface located in the WDIPSREUB.IDL file.

Figure D–1 shows the PSREUB API interfaces.

Figure D–1 PSREUB API Interfaces



WDIRoot Interface

Table D–2 lists the operations available in the WDIRoot interface of the **WDIPSRBIL.IDL** file.

Table D–2 PSREUB API WDIRoot Interface Operations

Operations	Description
connect	Returns a reference object to the WDIManager interface.
disconnect	Terminates the connection.

WDIManager Interface

Table D–3 lists the operations available in the WDIManager interface of the **WDIPSRBIL.IDL** file.

Table D–3 WDIManager Interface Operations

Operations	Description
startPSREUBSession destroyPSREUBSession	Start/destroy the PSREUBSession.
startSignal destroySignal	Start/destroy the WDISignal interface.
startInSignal destroyInSignal	Start/destroy the WDIInSignal interface.

PSREUBSession Interface Operations

Table D–4 lists the operations and the accompanying notifications available in the PSREUBSession interface of the **WDIPSREUB.IDL** file.

Table D–4 PSREUBSession Interface Operations

Operation	WDINotiication
exportCustomer	exportCustomerSucceeded operationFailed

Table D–4 (Cont.) PSREUBSession Interface Operations

Operation	WDINotiication
exportCustomer_v2	exportCustomerSucceeded_v2 operationFailed
exportOrder	exportOrderSucceeded operationFailed
exportOrder_v2	exportOrderSucceeded_v2 operationFailed
exportCustomerChange	exportCustomerChangeSucceeded operationFailed
exportCustomerChange_v2	exportCustomerChangeSucceeded_v2 operationFailed
setBillingInterfaceCode_v2	setBillingInterfaceCodeSucceeded_v2 operationFailed

Process Flows

This section describes business process flows associated with the PSREUB API. A third-party billing server should be designed to handle these process flows.

Process Flow for Send Bill Cust Gateway Event

1. The EventServer is running and polling the MetaSolv Solution database for gateway events in Sending status.
2. A new customer is created.
3. A new order is placed on that account and the SBC gateway event is initiated. This sets the gateway event status to Sending.
4. The EventServer picks up the gateway event and processes it. This process includes binding to the WDIRoot interface, connecting to WDIManager interface, and starting a session on the third-party billing server based on the Work Management gateway definition for PSRBilling.
5. The EventServer calls the *eventOccurred* operation on the third-party billing server passing in the parameter structure WDIEvent. The WDIEvent structure includes among other fields the document number and the gateway event name. The *eventOccured* operation must be written as part of the third-party server development.
6. The signal handler module on the third-party billing server activates the request handler module and updates the gateway event status to *In Progress*.
7. The third-party billing server gets the root, connects to the manager, and starts the session for the PSREUB server.
8. The third-party billing server gets the root, connects to the manager, and starts the session for the PSR server.
9. The third-party billing server calls the *exportCustomer_v2* operation on the PSREUB server, which returns the cust_acct_id based on the documentNumber.

10. The third-party billing server calls the *exportCustomerAccount_v2* operation on the PSR server, which returns all the customer account data as defined in PSRCustomerAccount structure in the WDIPSRTypes_v3.IDL file.
11. The third-party billing server imports the data into the billing system's database using the facilities provided by or supported by the billing system, such as an API, SQL, or an ODBC interface. To determine what facilities are provided by or supported by your billing system, and for information on how to use those features, see your billing system's documentation.
12. Depending on the success or failure of step 11, the signal handler module on the third-party billing server activates the request handler module and updates the gateway event status to Completed or Error.
13. Depending on the success or failure of step 12, the third-party billing server calls the *setBillingInterfaceCode_v2* operation on the PSREUB server. The successful call results in the update of the customer account billing interface code from N, meaning New, to A, meaning Accepted. This code should be left as is in the case of a failure.
14. The third-party billing server destroys the session and disconnects from the manager for both servers with which it is interfaced.

Process Flow for Send Bill Ord Gateway Event

1. The EventServer is up and running polling the MetaSolv Solution database for a gateway event status of Sending.
2. A new order is placed and the SBO gateway event is initiated. This sets the gateway event status to Sending.
3. The EventServer picks up the gateway event and processes it. This process includes getting the root, connecting to the manager and starting the session of the third-party billing server based on the Work Management gateway definition for PSRBilling.
4. The EventServer calls the *eventOccurred* operation on the third-party billing server passing in the WDIEvent parameter structure. This structure includes among other fields the document number and gateway event name. The *eventOccured* operation must be written as part of the third-party server development.
5. The signal handler module on the third-party billing server activates the request handler module and updates the gateway event status to In Progress.
6. The third-party billing server gets the root, connects to the manager and starts the session for the PSREUB server.
7. Based on the gateway event name of Send Bill Ord, the third-party billing server calls the *exportOrder_v2* operation on the PSREUB server. This call returns all the information in the Order structure as defined in the PSREUBTYPES_V2.IDL file.
8. The third-party billing server sends the billing information to billing system's database.
9. Depending on the success or failure of step 8, the signal handler module on the third-party billing server activates the request handler module and updates the gateway event status to Completed or Error.
10. The third-party billing server destroys the session and disconnects from the manager for the PSREUB server.

Process Flow for Customer Change Application Event

The process flow is for an application event, not a gateway event. The application event is initiated by clicking the OK button on the Customer Maintenance window for a customer who was previously sent to billing. The billing interface code for the customer must be A, meaning Accepted, or E for Error. If it is A, PSR will set the code to C, meaning Change, which is what invokes this interface point. If the billing interface code is E, then it has been sent to billing but something was wrong, and the user must change the information setting it back to C, so it will get resent to billing.

If the billing interface code for the customer is N, meaning New, the PSR module does not set the code to C because this would result in an attempt to update a customer that has not yet been sent to the billing system. Therefore, the PSR module only sets the billing interface code from A to C, not from N to C. Any changes made to a customer with a billing interface code of N are sent to the billing system through the Send Bill Cust gateway event.

1. The Event2Server is up and running polling the MetaSolv Solution database for a billing interface code C, meaning Change.
2. An existing account is updated on the Customer Maintenance window in MetaSolv Solution. The Customer Change Application Event is initiated only if the customer was previously added to the billing system's database. In other words only if the billing interface code is A, meaning Accepted or E, meaning Error.
3. When the OK button is clicked on the Customer Maintenance window in MetaSolv Solution, the client updates the billing interface code from A or E to C.
4. The Event2Server picks up the application event and processes it. This processing includes getting the root, connecting to the manager, and starting the session of the third-party billing server. Since this is not a gateway event, determination of the server is not based on the Work Management gateway definition for PSRBilling, it assumes the PSRBilling gateway definition.
5. The Event2Server calls the *eventOccurred* operation on the third-party billing server, passing in the WDIEvent parameter structure. The *eventOccured* operation must be written as part of the third-party billing server development.
6. The third-party billing server gets the root, connects to the manager, and starts the session for the PSREUB server.
7. The third-party billing server gets the root, connects to the manager, and starts the session for the PSR server.
8. The third-party billing server calls the *exportCustomerChange_v2* operation on the PSREUB server. This call returns the customer account ID based on the given document number.
9. The third-party billing server calls the *exportCustomerAccount_v2* operation on the PSR server, which returns the PSRCustomerAccount structure's information as defined in the WDIPSRTYPES_V3.IDL file.
10. The third-party billing server sends the information to the billing system's database.
11. Based on the success or failure of step 10, the third-party billing server calls the *setBillingInterfaceCd* operation on the PSREUB server, which updates the customer account billing interface code from C to A if successful. In the case of a failure the billing interface code should not be changed.
12. If there is anything wrong then the billing interface code should be updated to E for error.

13. The third-party billing server destroys the session and disconnects from the manager for both servers with which it is connected.

Viewing PSREUB API Event Errors in MetaSolv Solution

You can view errors for SBC and SBO gateway event on the MetaSolv Solution Work Queue Manager Window - Gateway Events Tab.

Errors for Customer Change Application Events can be viewed on the MetaSolv Solution Billing Discrepancies window. To open this window, select **Infrastructure> List> Interfaces> Billing Discrepancies** from the MetaSolv Solution main menu. The Billing Discrepancies window displays the error reported by the third-party billing server for that Customer Change Application Event.

See the online Help for more information.

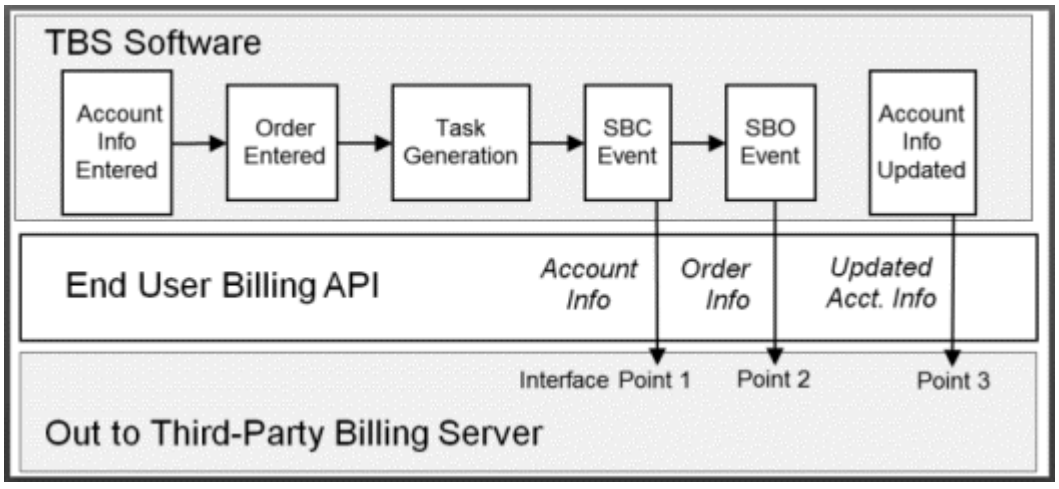
Note: The PSREUB API does not provide a persistence layer that the third-party billing server can use to update errors reported for Customer Change Application Events.

Solicited Messages

A solicited message is a message initiated by MetaSolv Solution. MetaSolv Solution plays the role of the client, and the third-party application plays the role of the server. The third-party application must use the IDL files provided with the PSR Order Entry API to implement the interfaces and operations shown in the following table:

Additional Process Flow Information

Figure D–2 PSREUB API Processing Flow Diagram



As shown in Figure D–2, account information and order details are entered and tasks for that order are generated in MetaSolv Solution. The PSREUB API passes this information to the third-party billing server at the SBC (Send Bill Cust) event, SBO (Send Bill Ord) event, and when account information is updated, which is also referred to as a Customer Change Application Event.

Interface Point 1: SBC Event

The PSREUB API enables account information to cross the interface at the SBC event to accomplish the following:

1. To avoid having an account established in a billing system that did not request service, and therefore would never bill.
2. To allow for pre-payments or deposits to be established on an account in a billing system before the order has been due-date completed in MetaSolv Solution.

Information Passed to the Server

The PSREUB API passes customer information. PSR account-level information that is passed to the server depends on the type of account. The product service request defines three types of accounts: customer accounts, billing accounts, and internal accounts.

Customer account information includes name, address, and contact information. Billing account information includes the same information plus bill cycle, tax exemptions, credit rating, auto payment information, and special handling codes. Internal account information is a name and a number, as internal accounts are used for departmental billing. Account information can be inserted, updated, and disconnected by MetaSolv Solution.

Interface Point 2: SBO Event

The PSREUB API sends order information to the third-party billing server at the completion of the SBO event. Users must select a provisioning plan that includes a task with the SBO event when the PSREUB API software option is enabled.

Information Passed to the Server

Order information passed to the third-party server from the PSREUB server includes products, services, features, and options, as well as the associated pricing, and customized attributes associated with the PSREUB API process points. Users must define products, services, features, and options as item types on the Product Specifications window. Item types are then grouped together as product offerings on the Product Catalog window. Product offerings, comprised of item types, are selected for an order. For each instance of an item type on an order, the product service request generates a unique service item ID. Every order can process New, Changed, Disconnect, Transfer Add, and Transfer Delete service items.

Implementation Concepts

MetaSolv Solution utilizes system-defined gateway events as well as an application event to notify the third-party billing server at specified points in the PSR workflow as shown in Figure D–2.

The gateway events used by the PSREUB API are:

- Send Bill Customer (SBC)
- Send Bill Order (SBO)

The SBC gateway event communicates information about a new customer account to the third-party billing server. This allows the third-party billing server to add the customer account to the billing system (see Interface Point 1 in Figure D–2, "PSREUB API Processing Flow Diagram"). The SBO gateway event communicates order

information to the third-party billing server. This allows order information to be added to the billing system.

Within the Work Management subsystem, the SBC event should always precede the SBO event.

> **WARNING: Both the SBC and SBO events must be completed before completion of the Due Date (DD) task. If you invoke SBC or SBO events after the service request is due date completed, and the third-party billing server reports an error; for example, "Invalid data" MetaSolv Solution does not allow you to supp the original service request to send the SBO or SBC again. This results in the data in the two systems being out of sync, and someone must manually change the data in one system to synchronize the data.**

The MetaSolv Solution client uses the Customer Change Application Event to notify the third-party billing server, through the Event2Server, when a customer account is modified after it was successfully sent to the billing server.

In order to interact successfully with the PSREUB API, your application must implement three major functions:

- A signal handler
- A request handler
- A response handler

The design you use when implementing these functions is entirely up to you. However, this documentation refers to the code that handles these functions as modules.

Signal Handler Module Design

The signal handler module implements the interfaces required to handle the SBC and SBO gateway events as well as the Customer Change Application Event. This module is also responsible for updating gateway event status to In Progress, Completed, or Error.

A WDIEvent data parameter structure is passed in.

Table D–5 lists the values of several fields for the SBC and SBO gateway events, and Customer Change Application Events.

Table D–5 SBC, SBO, and Customer Change Events

Event	*eventName* on WDIEvent	*documentNumber* on WDIEvent
SBC	"Send Bill Cust"	non-zero
SBO	"Send Bill Ord"	non-zero
Customer Change Application Event	"CCAE"	0 (zero)

Upon receiving a signal from the client, the signalhandler module activates the request handler module. For all types of events, the signal handler module is expected to return a WDIStatus data structure. In order to avoid locking up MetaSolv Solution, it is recommended that the signal handler return a status immediately upon activating the request handler module.

Request Handler Module Design

The request handler module addresses the functionality that the third-party billing server should implement to handle the information communicated by the client. Upon receiving event information from the signal handler, the request handler should perform the following steps:

1. Extract customer information (as applicable) by calling the *exportCustomerAccount* operation on the PSRSERVER.
2. Retrieve and examine the billing interface code before sending information to the billing system.

> **Note:** This step is essential because events may be initiated from MetaSolv Solution regardless of the value of the billing interface code. For example, new customer information needs to be sent to the billing system only if it has not already been sent. If the billing interface code indicates that it was successfully sent to the billing system, (status = A, meaning Accepted), then no information needs to be sent to the billing system and event status may be set to C, meaning Completed, or B, meaning Bypassed.

3. After data extraction and necessary conversions, the account information is passed to the billing system for processing. The UserPreference data structure received from the server contains user preference information that may be used by the third-party billing server. For example, user login information may be used to connect to the billing system.

Response Handler Module Design

The response handler module handles responses received from the billing system. The design of this module depends on factors such as the online/batch and synchronous/asynchronous nature of the interaction with the billing application.

Upon receiving a response from the billing system, the response handler module performs the necessary reverse translation/formatting using the formatting/translation module and then determines the operation's status. Based on the success or failure determination, this module should then:

1. Use the *setBillingInterfaceCode* operation in the PSREUBSession interface to update the billing interface code as applicable for the billing system used.
2. Record billing errors as applicable for the billing system used.
3. Update event status to Completed or Error.

Transaction Handling

Units of work in the third-party billing server should be carefully designed using the "all-or-nothing" principle. This is necessary so the billing system's database and the MetaSolv Solution database do not get out of sync.

PSR Service Item Vs. the Billing Service Instance

The PSREUB API does not process every service item in the same way. Instead, service items are interrogated to determine whether they are a service instance. Not every service item in a PSR is a service instance. Only service items that are service instances

are determined to be equipment. This determination is based on the definition of the item type on the Product Specifications window.

Each service item that is determined to be a service instance is further interrogated to determine the Usage Guide Key for the service instance. The *usage guide key* for the service instance is also based on the definition of the item type on the Product Specifications window.

The PSREUB API processes a service instance *usage guide key* as one of the following:

- Authorization code
- Auxiliary line
- Circuit ID
- Domain/Userid
- Tel Nbr/Auth Cd
- Telephone number
- Travel card number
- Universal

Pricing

Pricing can be associated with all service items. Each price associated with an item is interrogated and processed accordingly. Prices are based on type (recurring, non-recurring, usage) and level (account-level, service instance level). These price characteristics are defined on the Product Catalog window. The PSREUB API processes a service item price in PSR in one of five ways:

- Account level recurring charge
- Service instance level recurring charge
- Account level non-recurring charge
- Service instance level non-recurring charge
- Service instance level usage charge

Transfer of Products Between Customer Accounts

The PSR module and the PSR API allow transfer of designated service items between customer accounts. When a transfer order is entered in PSR the order is associated with the recipient customer account. These enhancements required corresponding updates to the PSREUB API.

For transfer PSRs, the donor customer account is the customer account from which the service items are being transferred, and the recipient customer account is the customer account to which the service items are being transferred.

A transfer PSR for a new customer (no previous service requests) has an order-level activity code of N, meaning New. A transfer PSR for an existing customer (any previous service requests) has an order-level activity code of C, meaning Changed. When service items are transferred, each transferred service item must be deleted from the donor customer account and added to the recipient customer account.

From a provisioning point of view, these services are not actually disconnected and added, because the service itself continues to be available; it just belongs to a different customer account. However, from a billing point of view the pricing associated with

these services is disconnected for the donor customer account and added for the recipient customer account. In other words the service itself remains uninterrupted or unchanged, only the account that pays for the service changes.

Using the ELEMENT, CONNECTOR, SYSTEM and PRDBUNDLE Item Types

If a service item of type ELEMENT, CONNECTOR, SYSTEM, or PRDBUNDLE has custom attributes, the PSREUB API passes the custom attributes. The structure that houses custom attribute information is defined in PSREUBTYPES_V2.IDL. The CustomAttribute structure itself is defined in WDIUTILS.IDL.

Items of type ELEMENT, SYSTEM, and PRDBUNDLE are never defined as billing service instances and therefore never have guide-to information. Items of type CONNECTOR are always defined as a billing service instance and therefore always have guide-to information. The usage guiding key for the CONNECTOR item type is always Universal.

Items of type ELEMENT, SYSTEM, or PRDBUNDLE never have PRILOC or SECLOC information. Items of type CONNECTOR always have either a PRILOC, a SECLOC, or both.

Glossary

The following list contains definitions of Oracle Communications MetaSolv Solution API terms as they relate to MetaSolv Solution documentation:

Access Carrier Name Abbreviation (ACNA)

A three-character abbreviation assigned by Telcordia to each Interexchange Carrier (IXC) and listed in the Local Exchange Routing Guide (LERG).

This abbreviation represents the access customer name to which the exchange carrier renders the access bill.

Access Customer Terminal Location (ACTL)

The COMMON LANGUAGE Location Identifier (CLLI) code of the Inter-Local Access Transport Area (InterLATA) facility terminal location of the access customer providing service.

ACNA (Access Carrier Name Abbreviation)

A three-character abbreviation assigned by Telcordia to each Interexchange Carrier (IXC) and listed in the Local Exchange Routing Guide (LERG).

This abbreviation represents the access customer name to which the exchange carrier renders the access bill.

ACTL (Access Customer Terminal Location)

The COMMON LANGUAGE Location Identifier (CLLI) code of the Inter-Local Access Transport Area (InterLATA) facility terminal location of the access customer providing service.

AID (Access Identifier)

Identifies the port address on a piece of equipment within the network element identified by the target identifier (TID). In the MetaSolv Solution database, the AID information is stored as the concatenated node address for the port address to which the circuit is assigned.

API (Application Programming Interface)

Software that permits other applications to access a specific area of data in the MetaSolv Solution database.

Application Programming Interface (API)

Software that permits other applications to access a specific area of data in the MetaSolv Solution database.

asynchronous operations

Operations in which control returns to the invoking application before the operation is acted upon. The invoked application returns the results to the calling application through a callback mechanism after the operation has been completed.

Asynchronous Transfer Mode (ATM)

A high bandwidth, low delay, packet-like switching and multiplexing technique.

ATM (Asynchronous Transfer Mode)

A high bandwidth, low delay, packet-like switching and multiplexing technique.

backup

The hardware and software resources available to recover data after a degradation or failure of one or more system components.

A copy of computer data on an external storage modem, such as floppy disk or tape.

bandwidth

A term used in various areas of the telecommunications industry (such as with facilities, SONET, Frame Relay, and ATM). In a channelized environment, (such as with facilities and SONET), the circuit positions used in MetaSolv Solution act as the discrete means of providing "bandwidth." The term "allocation of bandwidth" is also used in the industry. In MetaSolv Solution, "bandwidth" refers to a virtual circuit being "allocated" to bandwidth circuits through the Bandwidth Allocation table based on bit rates of each circuit rather than by a specific number of circuit positions (such as channels).

bandwidth circuits

In PVC (Permanent Virtual Circuit), bandwidth circuits are circuits that have virtual circuits assigned to them and have allocated capacity based on the digital bit rate as opposed to the method of using a distinct number of circuit positions (channels).

batch processing

A mode of computer operation in which a complete program or set of instructions is carried out from start to finish without any intervention from a user. Batch processing is a highly efficient way of using computer resources, but it does not allow for any input while the batch is running, or any corrections in the event of a flaw in the program or a system failure. For these reasons, it is primarily used for CPU-intensive tasks that are well established and can run reliably without supervision, often at night or on weekends when other demands on the system are low.

CAB (Carrier Access Billing)

A system that bills Interexchange Carriers (IXCs) for access time and hardware purchases.

carrier

A company that provides communications circuits. There are two types of carriers: private and common. Private carriers are not regulated and they can refuse to provide you service. Common carriers are regulated and they cannot refuse to provide you service. Most carriers (for example, MCI, AT&T, and Sprint) are common carriers.

Carrier Access Billing (CAB)

A system that bills Interexchange Carriers (IXCs) for access time and hardware purchases.

CCNA (Customer Carrier Name Abbreviation)

A Telcordia-maintained industry-standard code used to identify access customers (for example, AT&T and MCI).

Cell Relay Service (CRS)

An asynchronous transfer mode (ATM) term; a carrier service which supports the receipt and transmission of ATM cells between end-users in compliance with ATM standards and implementation specifications.

CLEI (Common Language Equipment Identifier)

Codes assigned by Telcordia (formerly Bellcore) to provide a standard method of identifying telecommunications equipment in a uniform, feature-oriented language. The code is a text/barcode label on the front of all equipment installed at Regional Bell Operating Company (RBOC) facilities that facilitates inventory, maintenance, planning, investment tracking, and circuit maintenance processes. Suppliers of telecommunication equipment give Telcordia technical data on their equipment, and Telcordia assigns a CLEI code to that specific product.

CNAM

CNAM is an acronym for:

Call Name Database (Sprint)

Calling Name (Caller ID)

Class Calling Name Delivery (Telcordia)

CBP (Convergent Billing Platform)

Allows for the bundling of services, such as long distance, cellular, paging, and cable, together onto a single monthly invoice.

COM (COMbined file)

A combined file used by the ASR/ISI Gateway for transporting multiple types of files. A COM file may contain various combinations of ASR Response files and ASR Error files.

commit

The final step in the successful completion of a previously started database change. The commit saves any pending changes to the database.

Common Language Equipment Identifier (CLEI)

Codes assigned by Telcordia (formerly Bellcore) to provide a standard method of identifying telecommunications equipment in a uniform, feature-oriented language. The code is a text/barcode label on the front of all equipment installed at Regional Bell Operating Company (RBOC) facilities that facilitates inventory, maintenance, planning, investment tracking, and circuit maintenance processes. Suppliers of telecommunication equipment give Telcordia technical data on their equipment, and Telcordia assigns a CLEI code to that specific product.

Common Object Request Broker Architecture (CORBA)

A standard architecture that allows different applications to communicate and exchange commands and data.

A central element in CORBA is the Object Request Broker (ORB). An ORB makes it possible for a client object to make a server request without having to know where in a network the server object or component is located and exactly what its interfaces are.

Concatenate

To allocate contiguous bandwidth for transport of a payload associated with a "super-rate service." The set of bits in the payload is treated as a single entity, as opposed to being treated as separate bits, bytes or time slots. The payload, therefore, is accepted, multiplexed, switched, transported and delivered as a single, contiguous "chunk" of payload data.

Convergent Billing Platform (CBP)

Allows for the bundling of services, such as long distance, cellular, paging, and cable, together onto a single monthly invoice.

CORBA (Common Object Request Broker Architecture)

A standard architecture that allows different applications to communicate and exchange commands and data.

A central element in CORBA is the Object Request Broker (ORB). An ORB makes it possible for a client object to make a server request without having to know where in a network the server object or component is located and exactly what its interfaces are.

cross-connect

A way of connecting two objects together. Cross-connects may be hard-wired or software based. Hard-wired cross-connects are used to connect two pieces of equipment using a physical media. Software cross-connects represent the connections made within a network node. The software cross-connect determines how a circuit is connected through an intelligent network element.

CRS (Cell Relay Service)

An asynchronous transfer mode (ATM) term; a carrier service which supports the receipt and transmission of ATM cells between end-users in compliance with ATM standards and implementation specifications.

Customer Carrier Name Abbreviation (CCNA)

A Telcordia-maintained industry-standard code used to identify access customers (for example, AT&T and MCI).

DACS (Digital Access and Cross-Connect Systems)

AT&T's proprietary digital cross-connect system (DCS) product. DCS is a type of switching/multiplexing equipment that permits per-channel DS0 electronic cross-connects from one T1 transmission facility to another, directly from the DS1 signal. That is, the DCS allows the 24 DS0 channels in one T1 line to be distributed among any of the other T1 lines connected to the DCS, without requiring external cross-connects.

daemon

A program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive. The daemon program forwards the requests to other programs (or processes) as appropriate.

dedicated plant

Describes a method used to build a telephone company's facilities. It is used when designated equipment, cables, and cable pairs are to be connected specifically to other pieces of equipment or locations. Once those connections are made they are seldom changed.

Design Layout Report (DLR)

A form designed according to the Industry Support Interface (ISI) standard originated by the Ordering and Billing Forum (OBF) committee. This form contains pertinent technical information sent to the access customer for review to ensure that the appropriate design has been provided and for the recording of its contents for future circuit activities. For MetaSolv Solution, this entity type and its dependents are used to record when the DLR was issued and to make the necessary changes to defaulted ASR values.

Digital Access and Cross-Connect Systems (DACS)

AT&T's proprietary digital cross-connect system (DCS) product. DCS is a type of switching/multiplexing equipment that permits per-channel DS0 electronic cross-connects from one T1 transmission facility to another, directly from the DS1 signal. That is, the DCS allows the 24 DS0 channels in one T1 line to be distributed among any of the other T1 lines connected to the DCS, without requiring external cross-connects.

DLR (Design Layout Report)

A form designed according to the Industry Support Interface (ISI) standard originated by the Ordering and Billing Forum (OBF) committee. This form contains pertinent technical information sent to the access customer for review to ensure that the appropriate design has been provided and for the recording of its contents for future circuit activities. For MetaSolv Solution, this entity type and its dependents are used to record when the DLR was issued and to make the necessary changes to defaulted ASR values.

EC (exchange carrier)

A company providing telecommunication in a licensed area.

ECCKT (Exchange Carrier Circuit Identification)

An AP Circuit ID or multiple circuit Ids.

end user

A customer who uses (rather than provides) telecommunications services.

end user location

The terminating location of telephone services for residential and business customers.

equipment specs

Documents that identify the properties and functionality of a piece of hardware. Equipment Specs are limited to items relevant to the operation of a circuit, such as channel banks, channel units, VF equipment, switches, cards, and so on.

escalation

The process of elevating a trouble ticket and making the appropriate parties aware that the resolution of the ticket is not progressing as well as expected and that assistance may be needed.

escalation method

The type of outage that has prompted a trouble ticket.

event

In the scope of the MetaSolv Solution APIs, an event represents the occurrence of something in MetaSolv Solution or in a third-party application that is of significance to the gateway user.

Exchange Carrier (EC)

A company providing telecommunication in a licensed area.

Exchange Carrier Circuit Identification (ECCKT)

An AP Circuit ID or multiple circuit Ids.

facility

Any one of the elements of a physical telephone plant required to provide service (for example, a phone or data line, switching system, or cables and microwave radio transmission systems).

fault management

Detects, isolates, and corrects network faults. It is also one of five categories of network management defined by the ISO (International Standards Union).

fixed length records

A set of data records all having the same number of characters.

flow-through provisioning

The automating of the activation process used to remotely communicate with the equipment in the field through Work Management tasks. MetaSolv Solution itself can act as the Service Management Layer (SML) that sends commands to the Network Management Layer (NML) where the commands are non-vendor specific. The NML then passes these commands and translates them into vendor terms and communicates these to the specific Network Element (NE), which is the actual equipment in the field. Examples of Network Elements are C.O. switch, Digital Loop Carrier (DLC), SONET node, and Digital Cross-connect System (DCS). MetaSolv Solution may also serve as the NML.

FOC (Form Order Confirmation)

A form the Local Exchange Carrier (LEC) submits to the Interexchange Carrier (IXC) to indicate the date when they will install ordered circuits.

Form Order Confirmation (FOC)

A form the Local Exchange Carrier (LEC) submits to the Interexchange Carrier (IXC) to indicate the date when they will install ordered circuits.

frame relay

A telecommunication service designed for cost-efficient data transmission for intermittent traffic between local area networks (LANs) and between end-points in a wide area network (WAN).

header record

The portion of a message containing information that guides the message to the correct destination. The header includes the sender's address, the receiver's address, the precedence level, routing instructions, synchronization pulses, etc.

ICSC (Interexchange Customer Service Center)

The telephone company's primary point of contact for handling the service needs of all long distance carriers. This center is responsible for outlining, configuring, and installing basic service upon customer request.

IDL (Interface Definition Language)

A programming language that helps define interfaces. IDL is inherently object oriented in nature.

IFR (Interface Repository)

A component of ORB that provides persistent storage of the interface definitions, acting as an online database and managing and providing access to a collection of object definitions.

INI file

An application-specific file that contains information about the initial configuration of the application.

interconnection interface

Using an API, MetaSolv Solution can be tightly integrated with a customer's proprietary software through software developed by third-party vendors like TMForum Common Interconnection Gateway Platform (CIGP).

Interexchange Customer Service Center (ICSC)

The telephone company's primary point of contact for handling the service needs of all long distance carriers. This center is responsible for outlining, configuring, and installing basic service upon customer request.

interface

A mechanical or electrical link connecting two or more pieces of equipment. An interface allows an independent system to interact with the MetaSolv Solution product family.

In this guide, the term interface refers to the CORBA IDL interface that describes the operations the interface object supports in a distributed application. These IDL definitions provide the information needed by clients for accessing objects across a network.

interface architecture

The collection of APIs and gateway integration software produced by Oracle e to permit access to the MetaSolv Solution database.

Interface Definition Language (IDL)

A programming language that helps define interfaces. IDL is inherently object oriented in nature.

Interface Repository (IFR)

A component of ORB that provides persistent storage of the interface definitions, acting as an online database and managing and providing access to a collection of object definitions.

International Standards Organization (ISO)

An international standards-setting organization.

Internet Service Provider (ISP)

A company that provides individuals and other companies access to the Internet and other related services such as web site building and hosting.

ISO (International Standards Organization)

An international standards-setting organization.

ISP (Internet Service Provider)

A company that provides individuals and other companies access to the Internet and other related services such as web site building and hosting.

item types

Predefined types which can be used to build product specifications. Relationships between the item types are also predefined; the item types and relationships together are commonly called MetaSolv Solution Rules. MetaSolv Solution only allows product specifications to be built that follow MetaSolv Solution Rules. These rules allow specific processing to be applied to item types.

Java Database Connectivity (JDBC)

An application program interface (API) specification for connecting programs written in Java to the data in popular databases.

JDBC (Java Database Connectivity)

An application program interface (API) specification for connecting programs written in Java to the data in popular databases.

LATA (Local Access Transport Area)

One of 161 geographical areas in the United States within which a local telephone company may offer local or long distance telecommunications service.

The LATA identifies which exchange carrier or Interexchange Carrier (IXC) may provide service in a defined area.

LIDB (Line Information Database)

A service that provides customers the ability to query Access Provider (AP) databases to determine whether a:

Caller is the authorized user of a valid AP calling card

Particular telephone number can accept collect or third-party billed calls before transmitting any call

Line Information Database (LIDB)

A service that provides customers the ability to query Access Provider (AP) databases to determine whether a:

Caller is the authorized user of a valid AP calling card

Particular telephone number can accept collect or third-party billed calls before transmitting any call

LNP (Local Number Portability)

A circuit-switched network capability that allows an end user to change service providers without having to change telephone numbers.

Local Access Transport Area (LATA)

One of 161 geographical areas in the United States within which a local telephone company may offer local or long distance telecommunications service.

The LATA identifies which exchange carrier or Interexchange Carrier (IXC) may provide service in a defined area.

Local Number Portability (LNP)

A circuit-switched network capability that allows an end user to change service providers without having to change telephone numbers.

Local Service Ordering Guidelines (LSOG)

A standardized set of guidelines used for ordering various local services. The local service request (LSR) is the administrative form that must accompany any local service request. This type of service request is used in a local competition environment to order unbundled elements such as loop service, number portability, and loop service with number portability. The local service provider sends a LSR to the network service provider when the local service provider cannot fill the requirements of an end user from owned resources.

Local Service Request (LSR)

The type of service request used in a local competition environment to order unbundled elements such as loop service, number portability, and loop service with number portability. An LSR is sent by the local service provider to the network service provider when the local service provider cannot fill the requirements of an end user from owned resources.

location

A physical location that is of interest for equipment inventory purposes. This location may have a Telcordia CLLI, a location identifier that is not a CLLI code, or may simply be identified by a street address. Circuit Design creates an entry in network location for End User PRILOCs and SECLOCs if it does not exist. Network location is a supertype of locations. Subtypes of locations include CLLI locations, end user locations, or terminal locations.

LSOG (Local Service Ordering Guidelines)

A standardized set of guidelines used for ordering various local services. The local service request (LSR) is the administrative form that must accompany any local service request. This type of service request is used in a local competition environment to order unbundled elements such as loop service, number portability, and loop service with number portability. The local service provider sends a LSR to the network service provider when the local service provider cannot fill the requirements of an end user from owned resources.

LSR (Local Service Request)

The type of service request used in a local competition environment to order unbundled elements such as loop service, number portability, and loop service with number portability. An LSR is sent by the local service provider to the network service provider when the local service provider cannot fill the requirements of an end user from owned resources.

mapping

The process of associating each bit transmitted by a service into the SONET payload structure that carries the service. For example, mapping a DS1 service into a SONET VT1.5 associates each bit of the DS1 with a location in the VT1.5.

network

The interconnection of equipment and outside plant components designed to provide an infrastructure fabric of facilities to support the transport of circuits. Each component of the network (Facilities, Equipment, Plant, and TFC Networks) may stand alone in the individual circuit design/assignment process. Alternatively, the components of the network may be combined to facilitate the designing process by allowing one assignment to encompass many network components together.

network element

A system such as a switch or Digital Cross-connect System (DCS) or a single shelf such as an Add-Drop Multiplexer (ADM). Another type of network element is a Digital Loop Carrier (DLC).

network node

Maintains information on an intelligent network element that makes up a telecommunications facility network.

NPAC SMS (Number Portability Administration Center and Service Management System)

Assists in administering Local Number Portability (LNP).

OBF (Ordering and Billing Forum)

A subcommittee of the Exchange Carriers Standards Association (ECSA). This forum discusses operational ordering, provisioning, billing, and presubscription.

Object Management Group (OMG)

Formed in 1989 by a group of vendors for the purpose of creating a standard architecture for distributed objects (also known as components) in networks. The architecture that resulted is the Common Object Request Broker Architecture (CORBA).

Object Request Broker (ORB)

The programming that acts as a broker between a client request for a service from a distributed object or component and the completion of that request. Having ORB support in a network means that a client program can request a service without having to understand where the server is in a distributed network or exactly what the interface to the server program looks like. Components can find out about each other and exchange interface information as they are running.

OMG (Object Management Group)

Formed in 1989 by a group of vendors for the purpose of creating a standard architecture for distributed objects (also known as components) in networks. The architecture that resulted is the Common Object Request Broker Architecture (CORBA).

ORB (Object Request Broker)

The programming that acts as a broker between a client request for a service from a distributed object or component and the completion of that request. Having ORB

support in a network means that a client program can request a service without having to understand where the server is in a distributed network or exactly what the interface to the server program looks like. Components can find out about each other and exchange interface information as they are running.

Ordering and Billing Forum (OBF)

A subcommittee of the Exchange Carriers Standards Association (ECSA). This forum discusses operational ordering, provisioning, billing, and presubscription.

Packet Internet Groper (PING)

A program used to test whether a particular network destination on the Internet is online.

password

A word or string or characters recognized by automatic means, permitting a user access to a place or to protected storage, files, or input/output devices.

ping (Packet Internet Groper)

A program used to test whether a particular network destination on the Internet is online.

port address

Maintains information on an equipment's assignable ports for transmission purposes. These ports can be either physical or virtual as in the relationship with the circuit positions associated with virtual (ST or VT) facilities. Port addresses can be either physical or enabled by the physical, as in the relationship with the circuit positions associated with facilities.

The port address can also be identified with a node address used for assignment selection. Other information can be maintained specific to the properties of the port, such as whether the port is line or drop, or identified as east or west.

Product Service Request (PSR)

An order request for end user products provided by a LEC. End user products include local dialtone services such as business lines and residential lines.

provisioning

The process of accomplishing the physical work necessary to implement the activity requested on an order.

This normally includes the design and the activation processes. For an install of a circuit, this would typically involve Circuit Design in MetaSolv Solution (making assignments) and activating the circuit.

PSR (Product Service Request)

An order request for end user products provided by a LEC. End user products include local dialtone services such as business lines and residential lines.

rate code

Identifies the bit rate associated with a circuit, facility, or equipment. For example, DS0, DS1, or DS3.

repeat trouble

Trouble reported on a service item two or more times within a specific period.

rollback

The undoing of partly completed database changes when a database transaction has failed.

SBO (Send Bill Ord)

A gateway event which must be associated with a task in the provisioning plan assigned to the service request.

scripts

The APIs use SQL (Structured Query Language) script. A script is a program or sequence of instructions that is interpreted or carried out by another program rather than by the computer processor (as a compiled program is).

Send Bill Ord (SBO)

A gateway event which must be associated with a task in the provisioning plan assigned to the service request.

service bureau

A data processing center that does work for others.

service category

Identifies the class of cell relay service for the Permanent Virtual Circuit (PVC). This information is identified in both directions of the PVC to support asymmetrical virtual services.

service item

A specific instance of a product or service. For example, a telephone line.

signal

An artifact that communicates information about an event. The point of reference for the API documentation is the MetaSolv Solution product line. Therefore, when reading material about signals, the direction of the signal in relation to MetaSolv Solution determines whether it is an inbound or outbound signal. When MetaSolv Solution sends the signal, that signal is called an outbound signal. When MetaSolv Solution receives the signal, that signal is called an inbound signal.

solicited message

A message issued by MetaSolv Solution acting as a client to another vendor.

SONET (Synchronous Optical NETwork)

An optical interface standard that allows interworking of transmission products from multiple vendors. It is a family of fiber-optic transmission rates from 51.84 Mbps to 13.22 Gbps, created to provide the flexibility needed to transport many digital signals with different capacities, and to provide a standard from which manufacturers can design.

staging tables

A set of interim database tables used by the ASR/ISI gateway when processing access service request (ASR) files.

synchronous operations

An operation in which the invoking application gets the results of the operation immediately upon the return of the call. The receiver of the operation acts upon that operation and returns the results. No callback mechanism is used.

Synchronous Optical Network (SONET)

An optical interface standard that allows interworking of transmission products from multiple vendors. It is a family of fiber-optic transmission rates from 51.84 Mbps to 13.22 Gbps, created to provide the flexibility needed to transport many digital signals with different capacities, and to provide a standard from which manufacturers can design.

Target Identifier (TID)

Identifies a group of equipment associated as part of a system or network element. In MetaSolv Solution, the TID information is maintained on the Node tab of the Network Element Properties window.

third-party

Describes developers who write interfaces to the MetaSolv Solution APIs for Oracle customers, allowing customers to access specific areas of the MetaSolv Solution database.

TID (Target Identifier)

Identifies a group of equipment associated as part of a system or network element. In MetaSolv Solution, the TID information is maintained on the Node tab of the Network Element Properties window.

transmission rate

The bit rates associated with a circuit, facility, or equipment. For example, DS0, DS1, DS3, N/A etc.

trouble

Any cause that may lead to or contribute to an end-user perceiving a failure or degradation on the quality of service of a telecommunications service.

VCI (Virtual Circuit Identifier)

The part of the logical connection address on the ATM switch port where the physical NNI or UNI circuit terminates. The PVC may be assigned one VCI per physical circuit. The VCI will accompany the virtual path identifier (VPI) if the PVC Connection Type is Channel; it will not be used if the type is Path. In a combined identification, the two will be displayed as VPI/VCI.

virtual

A term that has been used in various areas of the telecommunications industry such as with SONET, Frame Relay, and ATM. In a SONET environment, MetaSolv Solution uses virtual facilities the identify SONET auto-built ST and VT facilities as virtual facilities because the Virtual Indicator on the Transmission Facility Circuit table. In the MetaSolv Solution SONET application, the virtual facilities are used to transport other signals such as DS3 and DS1 circuits. In Frame Relay and ATM, MetaSolv Solution has used the virtual term for the permanent virtual circuit (PVC). In MetaSolv Solution, therefore, a Virtual Facility is used in the realm of SONET auto-built STS and VT facilities and Virtual Circuit is used when referring to the Frame Relay or ATM PVC.

Virtual Circuit Identifier (VCI)

The part of the logical connection address on the ATM switch port where the physical NNI or UNI circuit terminates. The permanent virtual circuit (PVC) may be assigned one VCI per physical circuit. The VCI will accompany the virtual path identifier (VPI) if the PVC Connection Type is Channel; it will not be used if the type is Path. In a combined identification, the two will be displayed as VPI/VCI.

Virtual Path Identifier (VPI)

The logical connection address on the ATM switch port where the physical NNI or UNI circuit terminates. The permanent virtual circuit (PVC) may be assigned one VPI per physical circuit. The VPI will be accompanied by the virtual circuit identifier (VCI) if the PVC Connection Type is Channel; the VPI alone will be used if the type is Path. In a combined identification, the two will be displayed as VPI/VCI.

VPI (Virtual Path Identifier)

The logical connection address on the ATM switch port where the physical NNI or UNI circuit terminates. The permanent virtual circuit (PVC) may be assigned one VPI per physical circuit. The VPI will be accompanied by the virtual circuit identifier (VCI) if the PVC Connection Type is Channel; the VPI alone will be used if the type is Path. In a combined identification, the two will be displayed as VPI/VCI.

work queue

A collection place for tasks associated with a service request. There are two types of work queues: child (individual) and parent (group). A child work queue is, typically, set up for one person. A parent work queue is most often set up for a group, department, or someone responsible for managing task assignments.