Oracle® Communications ASAP

Developer's Guide

Release 7.4
F40783-01
April 2022

ORACLE

Oracle Communications ASAP Developer's Guide, Release 7.4

F40783-01

Copyright © 2005, 2022, Oracle and/or its affiliates.

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

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

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

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

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

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

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

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

Contents

2 ASAP Database Tables

3 Shared Libraries

4 Provisioning Interfaces

5 Downstream Interfaces

B Oracle Execution Examples

C C++ SRP API Template Design

D API and Other Configuration Changes

Preface

This guide provides an overview of Oracle Communications ASAP (ASAP) application and library architecture, and describes ASAP database tables, shared libraries, provisioning and downstream interfaces, and web services.

The content of this manual is restricted to C/C++ reference information. Java-based information on the Java-enabled NEP, the Java SRP (JSRP), and Java Management Extensions (JMX) can be found in the *ASAP Online Reference*.

Audience

This document is intended for developers, system integrators, and other individuals who implement ASAP.

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.

Diversity and Inclusion

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

1

Development Overview

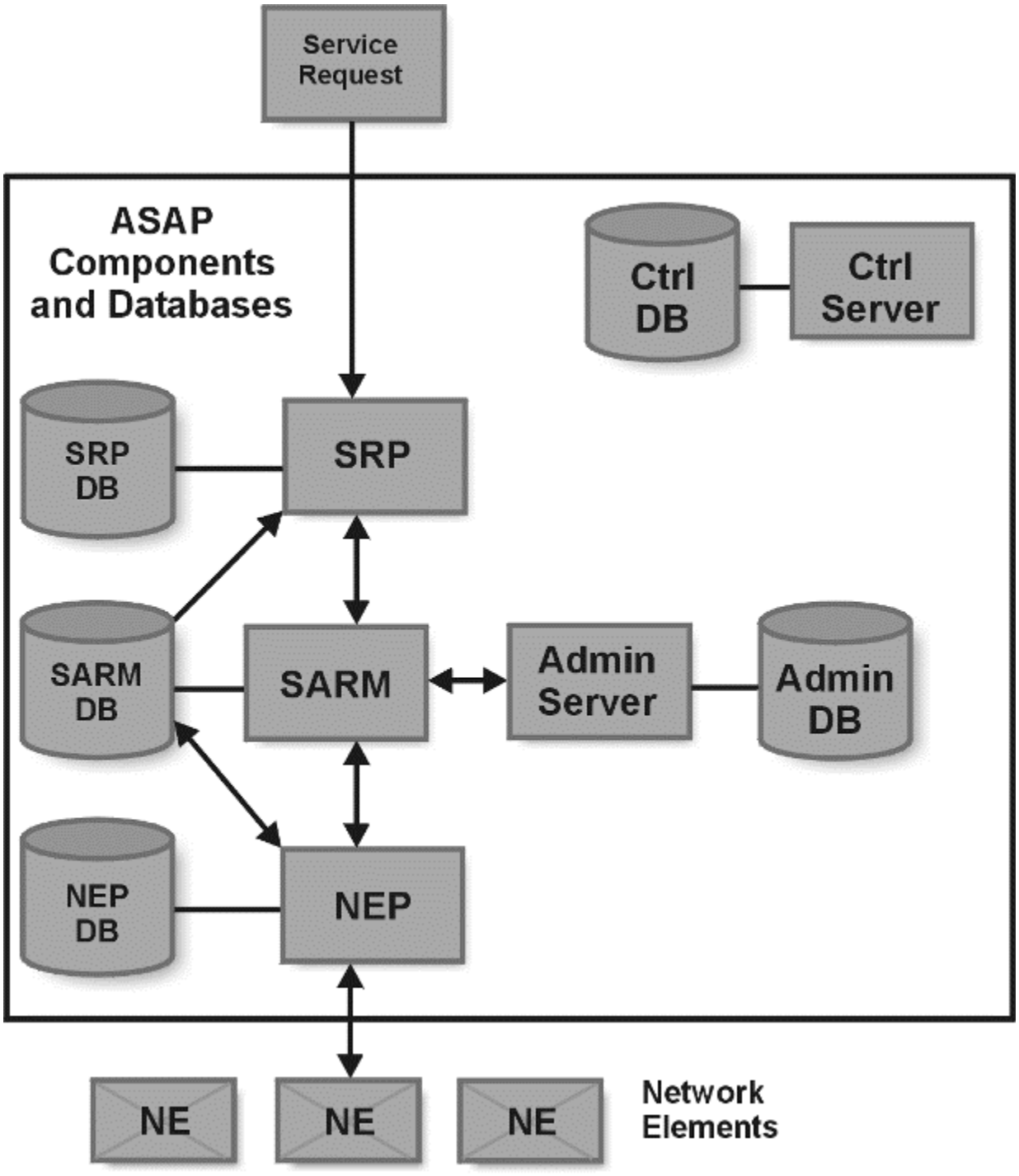
This chapter consists of the following sections:

Application architecture

Figure 1-1 outlines how ASAP components interrelate. This figure displays all ASAP application processes (for example, SRP and SARM) and their appropriate databases.

ASAP contains several processes and each process has its own database. Because of this close coupling of a process to its database, ASAP is distributed efficiently in a network environment.

Figure 1-1 ASAP Processes and Databases



The database engine:

ASAP uses the SQL Server as the database engine. A single SQL Server can contain one or more ASAP databases. In a completely distributed environment, each ASAP database may reside on a separate SQL Server, on a separate machine. This ability to distribute and scale the ASAP application transparently, is a fundamental feature of ASAP's design.

The ASAP control server:

The ASAP Control server manages ASAP's overall operation. The Control server:

- Starts and stops ASAP applications.
- Distributes ASAP across many machines.
- Maintains process and performance statistics about each application.
- Provides event notification, logging, alarming, and paging facilities that the applications use.
- Monitors the behavior of application clients and application servers.
- Issues system alarms to the proper alarm centers if an ASAP application terminates unexpectedly.

Client/server architecture

ASAP consists of a set of multithreaded UNIX Client/Server processes that communicate with each other and the associated database servers.

The Client/Server architecture has several advantages over traditional program architectures:

- Client/Server applications, such as ASAP, are easily distributed across several, possibly heterogeneous, platforms. The applications have a scalable architecture which you can expand upon to meet your future requirements.
- Application size and complexity is reduced significantly because common services are handled in a single location, by the server. This simplifies client applications, reduces duplicate code, and makes application maintenance easier.
- Client/Server architecture enables applications to be developed with distinct components. These components can be changed or replaced without affecting other parts of the application. Such components may be supplied as part of the base product or developed by individual customers to suit their own requirements.
- Client/Server architecture facilitates communication between varied applications. Client applications that use different communication protocols from the server cannot communicate directly with it; instead they must communicate through a “gateway" server that understands both protocols.

Multithreaded architecture

ASAP's multithreaded architecture makes more efficient use of the available resources than single-threaded architecture in the following ways:

You can combine several concurrent tasks into a single server process as threads of execution within that process. This means that there are fewer operating system processes running. Because of this, the operating system performs less process context-switching. *Thread* context-switching within a process is much more efficient than *process* context-switching performed by the operating system.

Since several tasks are running concurrently as threads within the server process, the threads communicate with one another through internal messages that use conventional process memory and semaphores as notification mechanisms. This method of communication is much more efficient than inter-process communication which would be required if each task was run as its own operating system process.

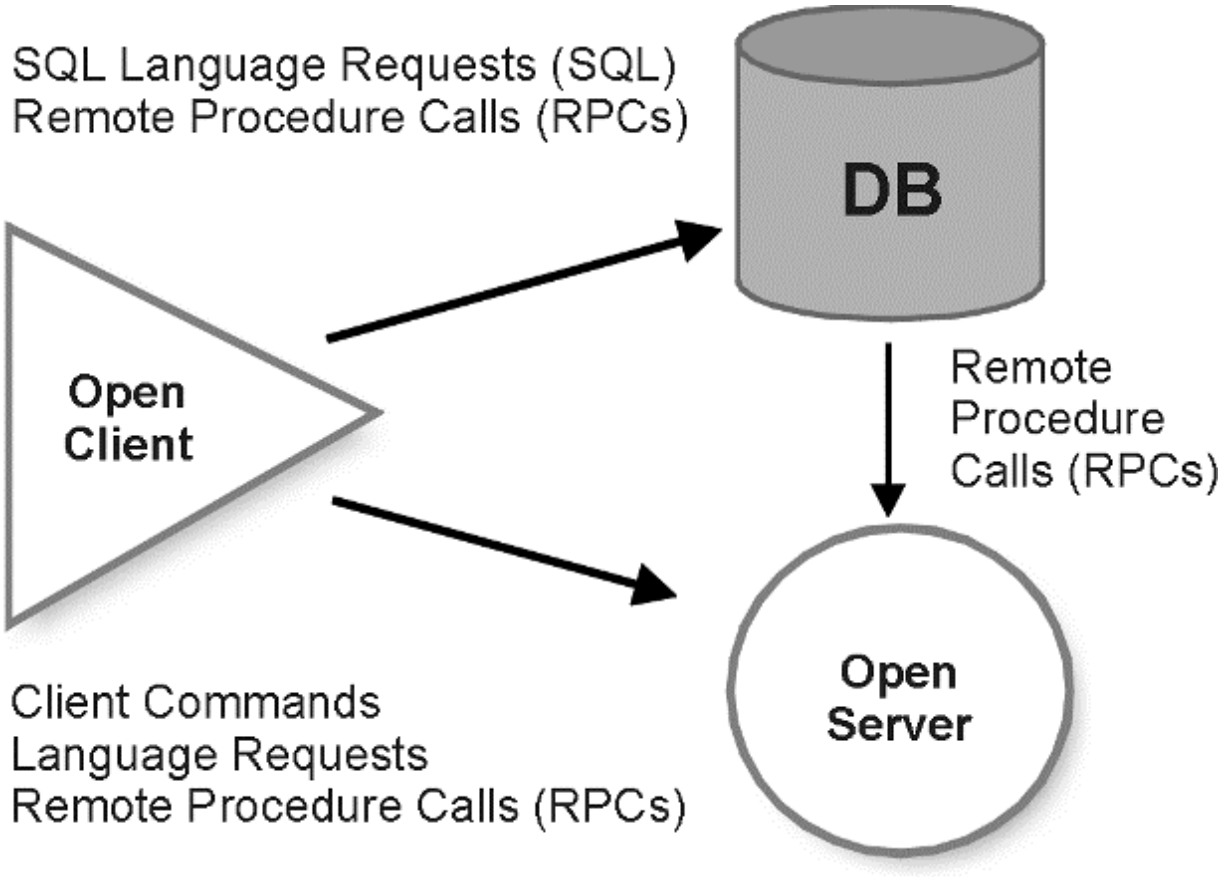
When the operating system allows a process to execute for a period of time on the CPU, the multithreaded server's internal scheduler schedules the various threads within the process over the duration of that period. Therefore, whenever a thread either explicitly yields the processor to another thread, or performs I/O activity (for example, disk and/or network access), the thread scheduler puts that thread to sleep and resumes the next thread in order of priority. As soon as the original thread's results are ready, the thread scheduler resumes that thread.

Due to the improved performance of multithreaded applications, most contemporary operating systems and database servers employ this architecture.

Open Server/Open Client ASAP components

ASAP is composed of three Client/Server systems: the RDBMS server, the Open server, and the Open client. Figure 1-2 outlines the Client/Server composition of ASAP and the methods of communication between components.

Figure 1-2 ASAP Client/Server Composition



RDBMS Server

The RDBMS Server is a stand-alone database engine. You can only add SQL definitions and data to the RDBMS Server.

Open Client

Open Client applications use the Open Client libraries (Client library and Database library) to communicate with the Open Server and the RDBMS Server. To communicate with these servers, the Open Client application uses either SQL or RPCs. The previous figure outlines how the Open Client communicates with the servers. In turn, the Open Server can receive results, messages, and notifications from the RDBMS Server and Open Client.

Open Server

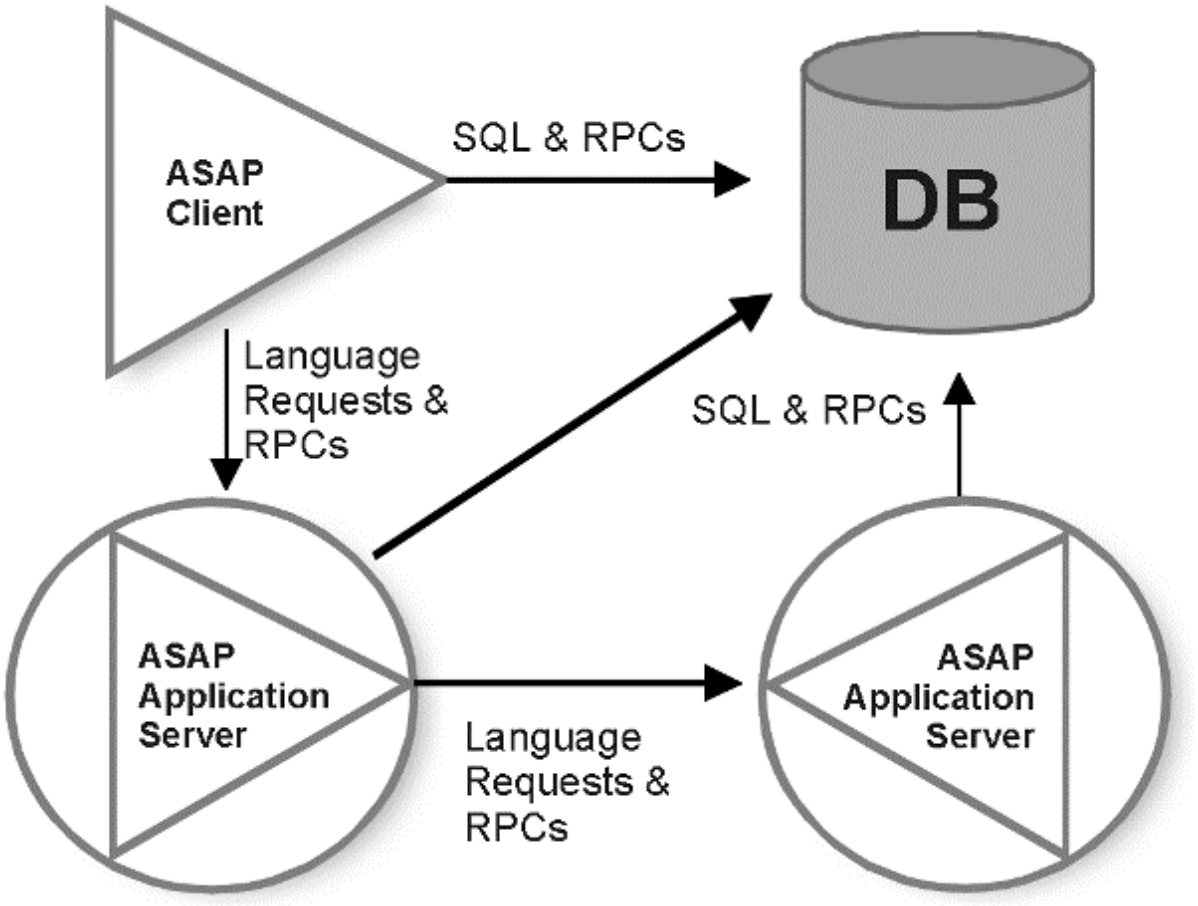
Open Server applications are built using the Open Server API and Open Client API. They are multithreaded processes containing both event driven threads (for example, handling a client connection) and background or service threads that perform non-event related tasks. They may receive language commands and RPCs from client processes, RPCs from RDBMS Servers, as well as send RPCs to other Open Servers (SARM to NEP) or RDBMS servers (access DB).

The Gateway Server application

A Gateway Server application is an application that incorporates both the client and server functionality. Such an application uses the client component to communicate with other servers and databases, and uses the server component to receive requests and RPCs from client processes and databases. ASAP consists of Gateway applications (for example, SARM, SRP, NEP) together with some client processes and databases. In this manual, every gateway application is referred to as an ASAP application server. The Client/Server configuration appears below.

The client API, provided by Sybase, co-exists in a multithreaded application process that is, it is fully re-entrant and does not invoke any blocking routines. In particular, the client API co-exists within a server application, resulting in a gateway application that possesses both client and server characteristics.

Figure 1-3 Gateway Server Application



Multithreaded environment

One of the principal issues to consider when using the ASAP API is the multithreaded environment that runs under UNIX. In UNIX, many processes run concurrently. Process execution, resource allocation, and CPU usage, for example, are controlled by the UNIX process scheduler.

A multithreaded application using the Open Server API, is one in which there are many Open Server execution threads within the UNIX process. All the Open Server threads share the resources with the host operating system process. The Open Server library that is linked to each Open Server application provides concurrence by periodically suspending the running thread and resuming another. This thread-context switch happens so quickly that the threads appear to run continuously from the point of view of the client application communicating with the Open Server.

Open Server thread scheduler

The Open Server library provides a specialized thread called the thread scheduler. The thread scheduler performs context switches between the threads in the server. A thread has an execution context that includes its stack and register environment. The

thread scheduler saves the execution context of the running thread, selects another thread to resume, restores its context, and runs it.

An Open Server application is a UNIX process whose execution is controlled by the UNIX process scheduler. The UNIX process scheduler context-switches the UNIX processes. Within the Open Server process, the thread scheduler context-switches the threads.

With context-switching, CPU resources are used more effectively. Any thread that performs I/O or other time-consuming operations is suspended and another thread runs while the first thread awaits the results of its operation.

Overall, the elements of the process now communicate extremely quickly. There is no data movement at all, only pointers passing within the UNIX process. Because the resources are managed more effectively, the application itself executes more quickly. Therefore, the application is a collection of threads within an Open Server, instead of a number of UNIX processes.

Open Server threads within the process have their own stack and register environments. The threads do, however, share the resources of the operating system process that is executing the server. For example, the standard input and standard output is the same for every thread in the server. If threads regularly write to the standard output, you must be careful not to mix the output of several threads on the standard output.

Programming in a multithreaded environment

Consider the following when programming in a multithreaded environment.

- Make all code re-entrant – Open Server threads execute the same code image, so the program must not modify itself. If a thread resumes code that was changed while the thread was suspended, the results are difficult to predict and likely to be unrecoverable. In a multithreaded environment, you must take into account the possibility of concurrence within a single UNIX process and not only between UNIX processes.

  For more information about Mutexes, refer to the subsection “Mutually Exclusive Semaphores (Mutexes)" in the section “Inter-Thread Communication" of this chapter.

- Protect shared resources – Make sure to protect shared resources such as global data, file descriptors, devices, and so on. While updating a shared global data item, do not call a routine that could suspend the thread unless you have taken steps to prevent other threads from accessing the data (such as a mutex). Otherwise, another thread could be working with inconsistent data. Watch for program logic that assumes it has sole access to a resource.

- Avoid static variables in routines that could be executed by more than one thread – If a thread accesses a static variable that might be accessed by other threads, the variable's value may have been changed since it was last referenced by that thread. It is safer to use automatic variables because each thread has its own stack. When static variables must be used, protect access to them by using the technique for protecting variables as shared resources, outlined in the previous point.

- Be careful with certain UNIX system calls and C library routines – The standard C library was not written to handle a multithreaded process explicitly. Therefore, certain functions may maintain a global variable invisible to you in order to save some information internal to the function. An example is **strtok()**. After **strtok()** makes its initial call, it maintains the pointer to its current position within the string. If two threads are using **strtok()** at the same time, it is possible that one of the **strtok()** calls on one string could return the address of some part of the other string, or worse, crash the server.

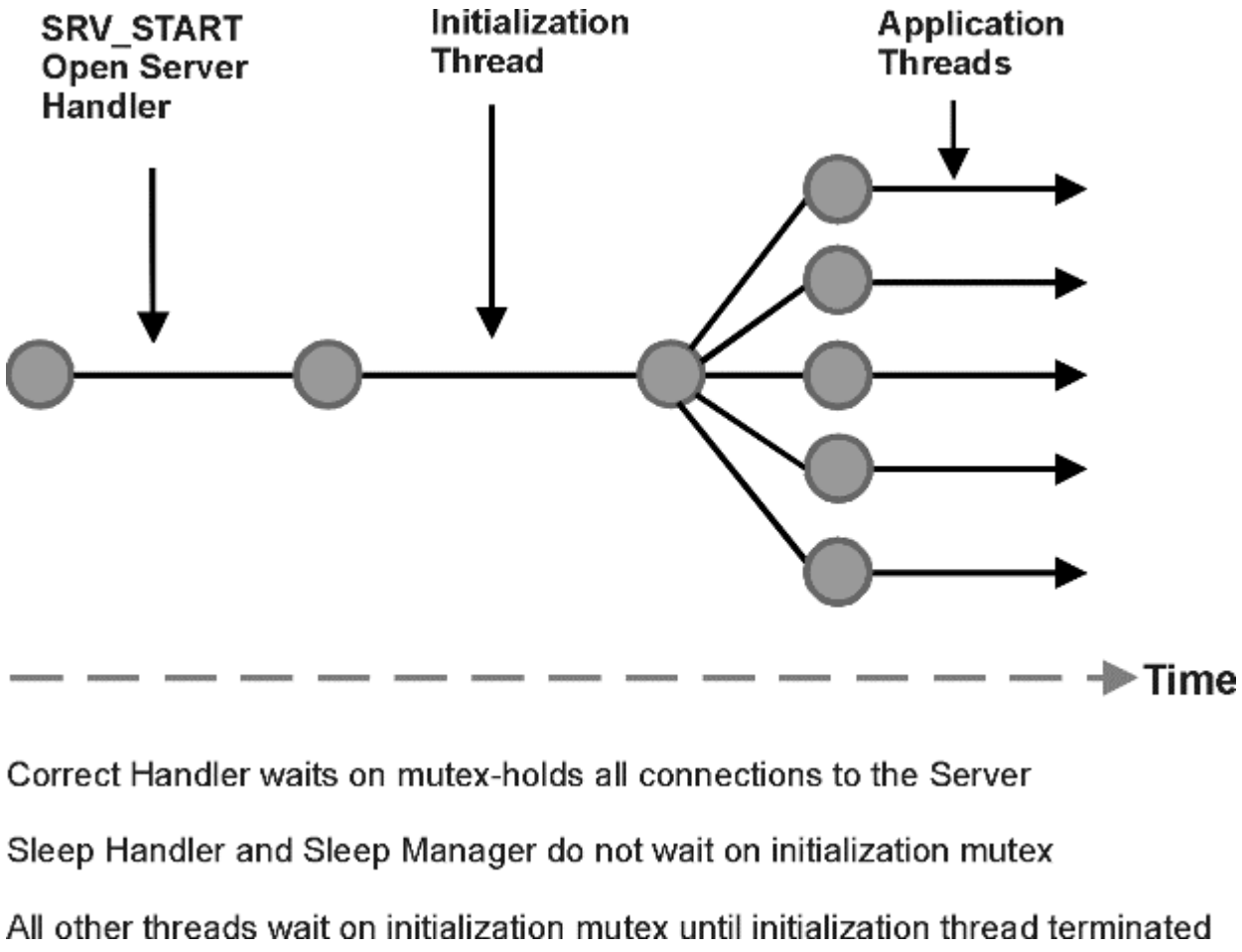
- Avoid blocking UNIX system calls – UNIX system calls such as **sleep()** and **read()** can block the entire UNIX process, not just the individual thread. Generally, the Open Server API and the ASAP API provide non-blocking versions of these system calls.
- Be aware of linking in third-party libraries – Some third party libraries may include blocking calls within them that could affect the Open Server operation. Examples of such libraries include LU6.2 API libraries and X.25 libraries.
- Be aware of the thread stack size – Each thread within an Open Server has its own stack environment. If a particular thread exceeds its stack size, the server terminates. Therefore, do not use recursive routines that may result in excessive stack depth or stack size, such as unbalanced binary trees, large automatic stack objects, and so on.

Multithreaded procedural server initialization

In the initialization of the ASAP application server, the ASAP API installs a **SRV_START** Open Server event handler which is called by the Open Server library to start the server process. The server process is single-threaded while the **SRV_START** handler is executing. The server can spawn service threads, create message queues and mutexes, among other things, from within this start handler. Such spawned threads do not begin executing until the handler has finished.

It is not possible to send or receive thread messages, to lock mutexes or to perform any network input/output operations within this handler.

Figure 1-4 Multithreaded Procedural Server Initialization



During this handler execution, the application supplied **appl_initialize()** API routine is called. In **appl_initialize()**, the application registers one or more initialization routines, by means of the **ASC_reg_init_func()** routine, to be executed before the application threads start up. Such routines generally load static data from the database into memory tables, and therefore cannot be called directly from **appl_initialize()** since it is part of the **SRV_START** handler.

In order to control the multithreaded server initialization, the ASAP API initially locks an initialization mutex and spawns an initialization thread which is allowed to run when the **SRV_START** handler finishes. All other application threads (except the Sleep Handler and Sleep Wakeup) are blocked by waiting on an initialization mutex. This includes all connections received by this application server.

The initialization thread processes each registered initialization routine in series. Once all such routines have been completed, the initialization thread unlocks the initialization mutex, allowing the application threads to begin execution with the assurance that all the necessary initialization has been performed.

Inter-process communication

The Open Server and Open Client APIs provide inter-process communication facilities that are used by ASAP applications. The principal inter-process communication facilities provided by APIs are RPCs, registered procedures, and language requests.

To handle communications between servers, you are provided with application server driver threads.

RPCs and registered procedures

The RPC/Registered Procedure mechanism is the predominant inter-process communication mechanism used in ASAP.

A Remote Procedure Call (RPC) is a function that is executed in an RDBMS Server or a “C" function that is executed in an application server by an application client or server process. To execute the procedure, the sending server or client first establishes a network connection with the destination server and passes the appropriate information to the destination server in name-value pair format. The sending server then executes the procedure or function, waits for data rows, returns status or text messages to the receiving server, and, finally, ends the execution of the procedure.

Registered procedures provide more efficient processing than RPCs but do not allow options in the procedure call. Registered procedures do, however, give servers the ability to notify clients whenever a registered procedure executes.

This method of communication is fast and efficient, especially when a network connection is already established for the sender to transmit its RPC/Registered Procedure and receive the associated results.

Executing functions:

The following procedures describes how to execute functions in Oracle.

To execute a function in Oracle:

1. Log in to the database by typing:

   ```
   sqlplus $<server_user>/$<server_password>
   ```

   <*server_user*> – is your user name for logging into the server <*server_password*> – is your password

2. Execute the function by typing:

   ```
   variable retval number;
   exec :retval :=<stored_procedure> (<parameter_name(s)>)
   ;
   ```

   <*stored_procedure*> – is the name of the function and <*parameter_name(s)*> – is the name of each column identified in the applicable database table

Language requests

To facilitate larger data volume transfers, the sending application client or server transmits a language request to the receiving server process. For these large data volumes, language requests are more efficient than RPCs.

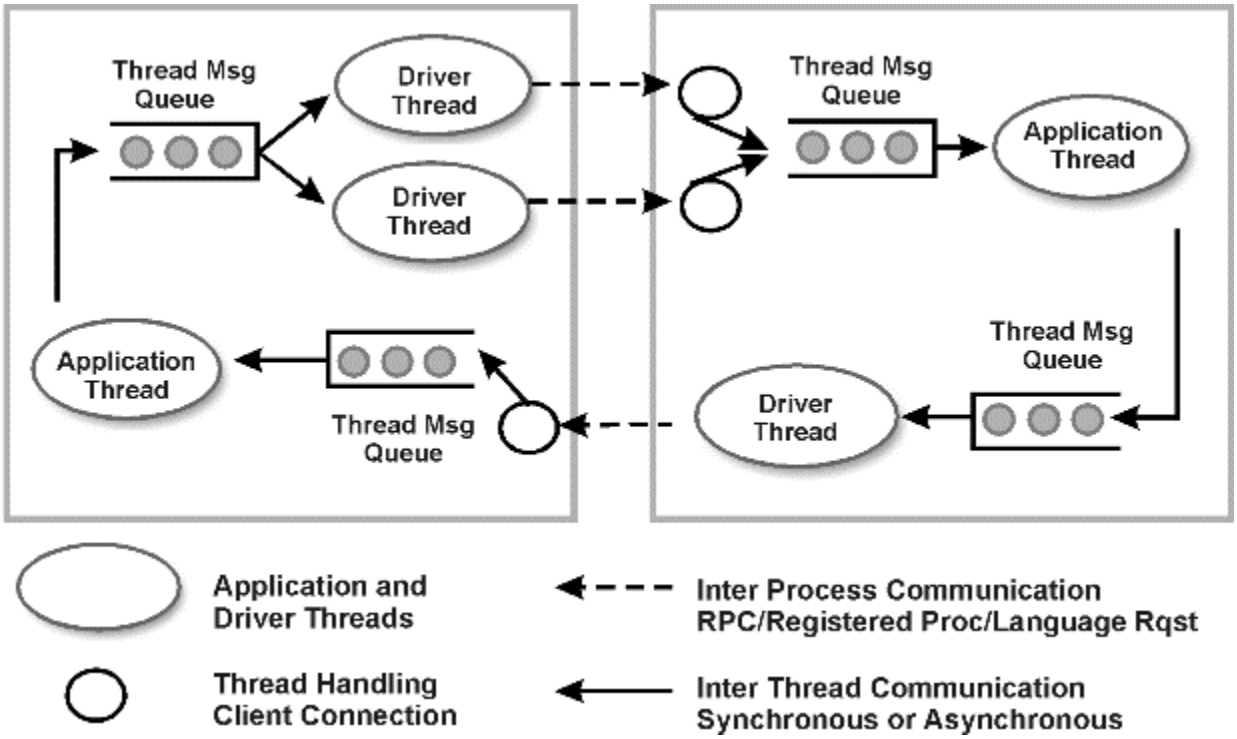
Application server driver threads

In most application servers that communicate with other application servers, a special thread called a driver thread is spawned on the sender side. This driver thread handles the communication between the servers. Any thread within the server that wishes to communicate with the other server interfaces with the driver thread and leaves the

driver thread to perform the actual RPC/Registered Procedure call. If the call fails, the driver thread marks the receiving server as down.

Figure 1-5 illustrates that on the receiver side, an RPC/Registered Procedure handler routine interprets the incoming RPC/Registered Procedure and takes the appropriate action.

Figure 1-5 Application Server Driver Threads



If a driver thread has been idle for a period of time (for example, one minute), it checks the integrity of the RPC connection just in case the other server process has died. To do so, the driver executes a “heartbeat" RPC called **kick_start**.

The heartbeat RPC checks the integrity of the network connection. If the connection is down, the driver thread closes the network connection. The driver thread's ability to close the network connection ensures that the application server will start up again. Another detection technique uses callbacks to trigger, when the connection between applications is broken.

If the driver thread does not close the network connection, the driver thread will not release its end of the connection and the UNIX kernel will not free up the socket associated with the network connection. Consequently, when the other server tries to start up and listen on its master network port, it will be unable to do so because the socket upon which it is supposed to listen is already in operation (in other words, it has not yet been released). Therefore, the other application server will not be able to start up.

All driver threads within ASAP employ this mechanism to validate the network connections. ASAP assumes that the RDBMS Server will not terminate unexpectedly and, therefore, does not check network connections to the RDBMS Server.

Inter-thread communication

Within each multithreaded application server, there might be many threads of execution and these threads may need to communicate with each other. The APIs provide a number of tools to allow for this “inter-thread" communication.

Mutually exclusive semaphores (mutexes)

A Mutual Exclusion Semaphore (mutex) is a logical object, much like a semaphore under UNIX. The Open Server API allows the mutex to lock only one thread.

Threads that share global variables, structures, tables, and so forth, may only access these resources to read or update if the threads first lock the mutex associated with them. Once the mutex is locked, only one thread has exclusive access to a resource. Other threads waiting to access that resource can only do so when the current thread is finished using it. Once that thread is finished using the resource, it unlocks the mutex, and grants access to the next thread in line. The new thread must lock the mutex again before accessing the resource. A thread may block waiting on a mutex.

Mutexes are primarily used to protect resources from multiple concurrent updates by more than one thread. Therefore, if there is a resource within an application server that could be updated by more than one thread at a time, you should protect it using a mutex. Such protection is generally encapsulated in a single function which manages the mutex updates.

Thread message queues

Message queues let threads communicate with each other and are often used to send data to other threads within an application server.

The message itself resides in memory shared by the sending and receiving threads. The thread that puts the message into a queue and the thread that reads it must agree on the message format. Be careful that the sending thread does not overwrite the message memory address before the reading thread receives the message.

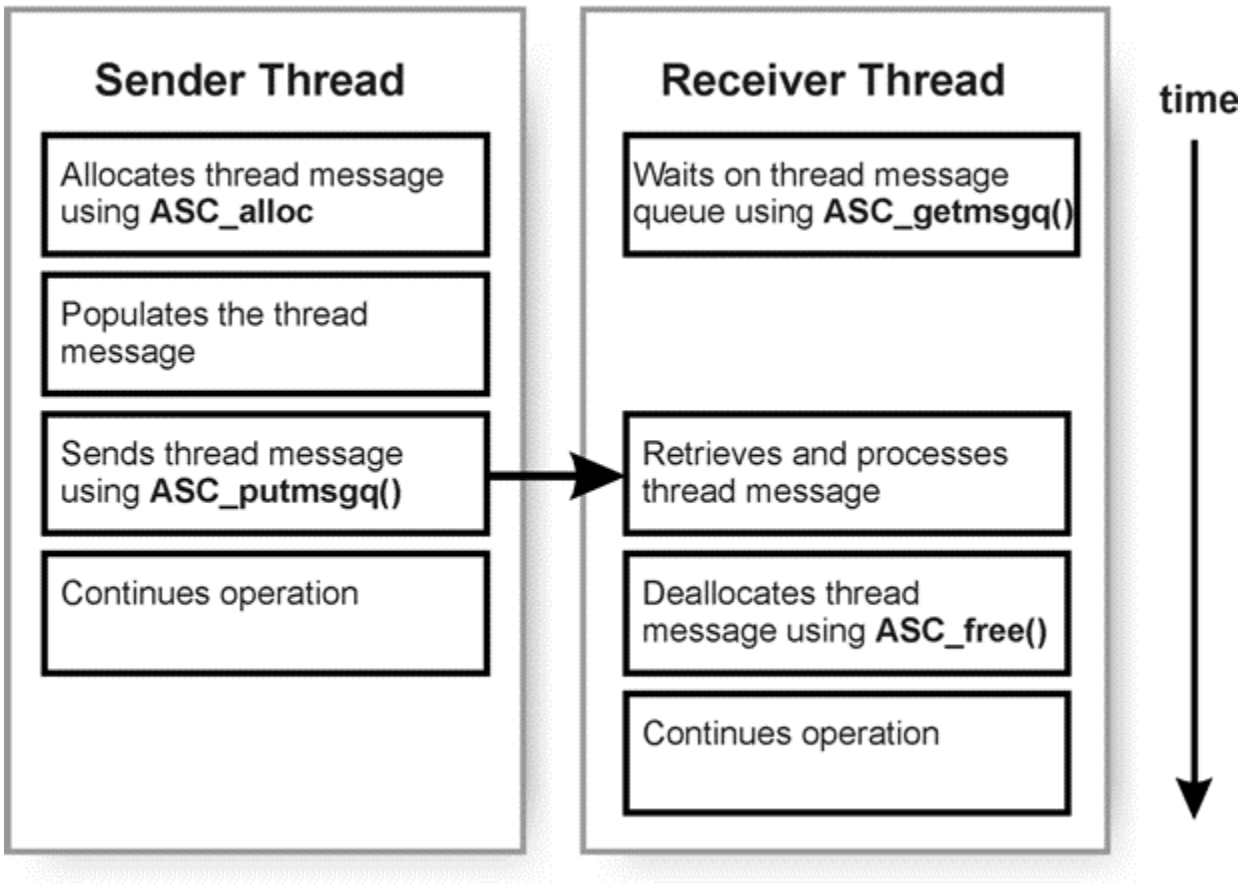
There are two main types of inter-thread messages employed within ASAP: asynchronous thread messages and synchronous thread messages. These two types of messages are explained in the following subsections.

Asynchronous thread messages:

Asynchronous thread messages are the most common types of messages passed between threads in ASAP. With these messages, the sender allocates memory for the message structure, fills in the message details, sends the message to the receiver, and continues normal operation.

The receiver receives the message, takes the appropriate action on its contents, and then frees the memory area allocated to the message. There is no needed synchronization between the sending and receiving threads. The only prerequisite for this type of communication is that the receiver must have a thread message queue. Figure 1-6 illustrates this process flow.

Figure 1-6 Asynchronous Thread Messages



Synchronous thread messages:

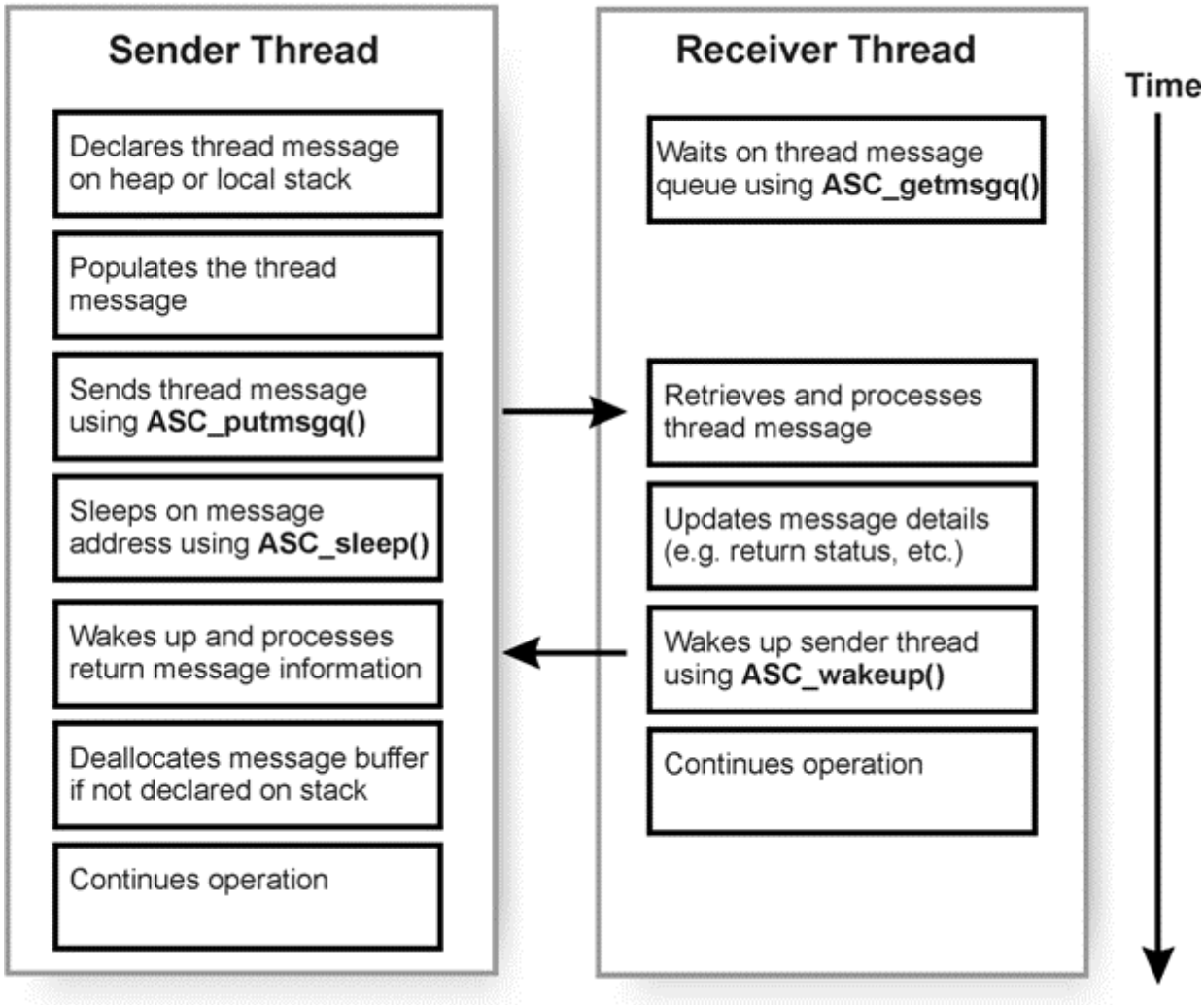
In some cases, the sender may need to wait before sending the message. For instance, it may need to wait on a status field entered in the message by the receiver. When this happens, the sender must wait until the receiver updates the relevant information in the message before referencing it. In this case, the sender usually declares the message on its stack, fills in the message details, and sends the message to the receiver.

Immediately after sending the message to the receiver, the sender thread goes to sleep and only wakes up when the receiver updates the message and explicitly wakes up the sender. When the receiver gets the message, it reads it and may update certain fields before waking up the thread that was sleeping on this message. At this point, both the sender and receiver continue with normal operations. This process flow appears in the following illustration.

Use the synchronous message method if a message sender requires return parameters or a message status.

> **Note:**
>
> Both the sending and receiving threads must agree on both the message format and the message type (synchronous or asynchronous). If the sender sends an asynchronous message and the receiver expects a synchronous one, then the memory allocated for the message will never be freed. Conversely, if the sender sends a synchronous message and the receiver expects an asynchronous one, the sender will stay asleep and the receiver will try to free the message on the sender's stack. This will lead to unpredictable results.

Figure 1-7 Synchronous Thread Messages



Functions and configuration variables:

There are four functions that must be used when dealing with thread message queues within ASAP: **ASC_createmsgq**, **ASC_deletemsgq**, **ASC_putmsgq**, and

ASC_getmsgq. The use of these functions must be consistent throughout the application.

Message queue statistics may also be dumped to a file.

System monitoring tool:

The ASAP Utility Script (asap_utils) is a menu that provides access from UNIX to a set of monitoring utilities for ASAP. You can access sysmon through the Real-time System Monitoring option (109) of the asap_utils menu. You can also monitor multiple servers at the same time by selecting the Real-time System Monitoring option (109) again. Once the data collection time period has passed, sysmon output files will be created in the ASAP systems diagnostic file directory.

> **Note:**
>
> Option 101, View Server Msg Queue Statistics, available in previous versions of ASAP, as well as the configuration parameter DIAG_MSGQUEUES and the RPC diag_msgqueues, have been replaced by the functionality available from option 109, Real-time System Monitoring.
>
> The system monitoring tool is not available to C++ SRPs.

Sample message queue statistics

```
----------------------
Tuning - Message Queue
----------------------
Description            Count   Total      Min   Max    Average  Mean Deviation
------------           ------  ------     ----  -----  -------  ---- ---------
ASDL Provision Queue
message read wait time  1056   5995.5     0.9   62.2     5.7    15.1    10.2
messages sent (count)   1056
queue idle-time (ms)    1056   5971905.0  0.0   114374.2 5655.2 23775.0 19062.4
queue size (count)      1056   0.0        0.0   0.0      0.0    0.0     0.0
...

Group Manager Msg Q
message read wait time  2469   38896.8    2.5   290.5    15.8   61.5    48.0
messages sent (count)   2469
queue idle-time (ms)    2469  18707440.6 0.0  71880.3  7576.9 18294.2 119980.0
queue size (count)      2469   27184.0     0.0   27.1     7.2    1.5     4.5
...
```

Device-oriented threads and socketpair messaging

Device-oriented threads typically watch for input from one or more devices using the **ASC_poll()** API call. Such a thread cannot wait on the internal message queue to receive inter-thread messages from other threads. This thread will create a socketpair and publish one end of the socketpair as the socket device to which the other threads can write messages. The thread adds the other end of the socketpair to the list of devices that are polled for input. If any thread writes to the write-endpoint of the socketpair, the device will detect it via **ASC_poll()** and then read the message using the read-endpoint. Only asynchronous messages are supported using socketpair messaging.

Both the sending and receiving threads must agree on the format of the message.

This messaging technique is used internally in the ASAP core development. There are no explicit API calls provided as this is an internal feature of the ASAP API. Socketpair messaging is commonly used in applications interfacing with other external systems.

Notes for C++ compilation

The UNUSED macro:

This macro takes a single argument and is used in function calls where one of the arguments is unused. It prevents the C++ compiler from complaining about the unused argument. Refer to the example below.

```
CS_INT example_func(int arg1, int arg2, int UNUSED(arg3))
{

CS_INT arg4;
.
.
.
arg1 ++;
arg4 = arg2 + arg1;
.
.
.
return arg4;

}
```

In this example, the C++ compiler will accept the omission of arg3 in example_func.

Library architecture

This chapter outlines ASAP's library architecture. The following sections are included in this chapter:

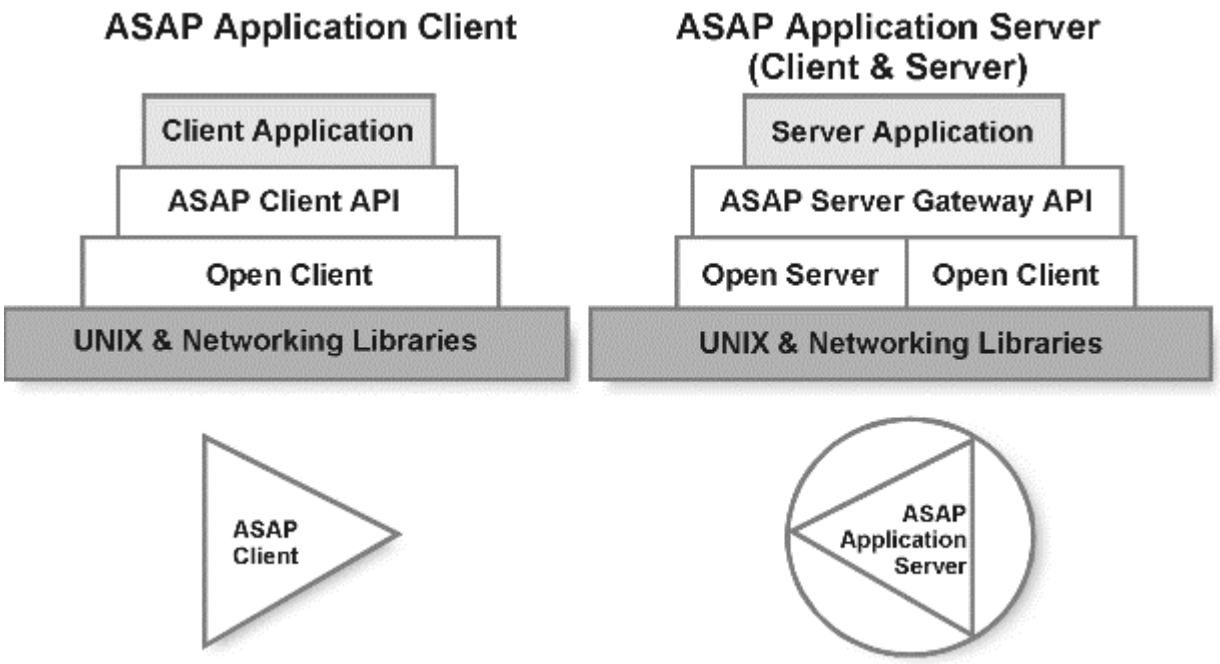
- ASAP API Development Structure
- API Library Structures
- ASAP API Application Development
- SRP Server Application Structure
- Generic NEP Application Structure
- Multi-Protocol NEP Structure

ASAP API development structure

The ASAP development API builds upon the Open Client and Open Server libraries. ASAP client applications use the ASAP Client API which interfaces with the Open Client. ASAP server applications use the ASAP Server API to communicate with both the Open Client and Open Server libraries.

Figure 1-8 summarizes the API hierarchy.

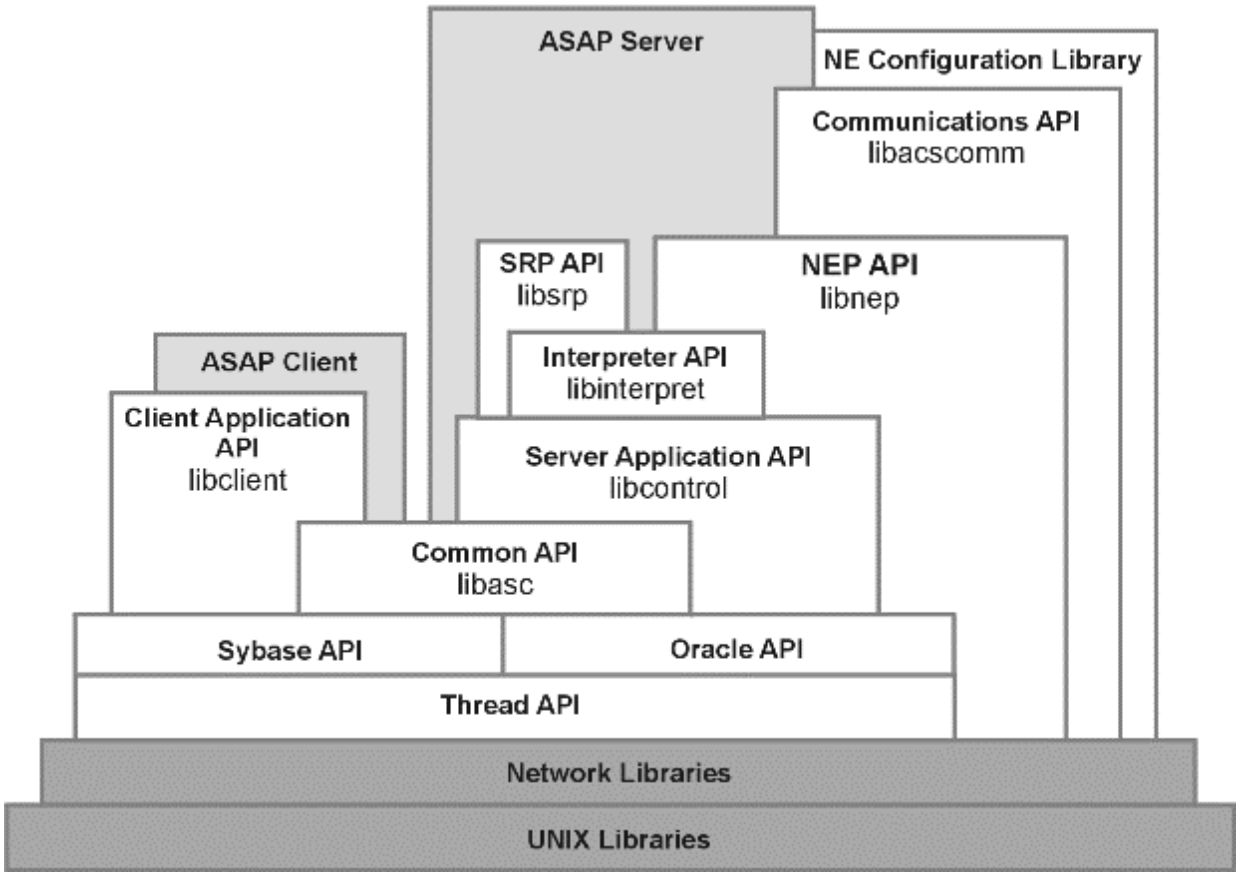
Figure 1-8 ASAP API Hierarchy



API library structures

A detailed API structure emphasizing the application APIs appears in Figure 1-9. This figure illustrates the APIs that can be linked to an ASAP application. In general, each application links to a subset of the outlined libraries. For more information about each of the API libraries, refer to the subsections that follow.

Figure 1-9 API Library Structures



Common API library – libasc

The Common API library, **libasc**, provides a set of API routines common to both the client and server application libraries: libclient and libcontrol. This library is linked to both application clients and servers.

The libasc library provides you with considerable functionality, including:

- Diagnostic routines common to both clients and servers
- System event generation
- Application configuration parameter determination
- Network connection management
- Performance parameter generation
- Registered Procedures API
- Remote Procedure Calls API

Client application API library – libclient

The Client Application API library, **libclient**, has an application client template to which you can add application-dependent functionality. This library, which is employed by every ASAP application client, provides routines specific to client applications.

The primary advantage of **libclient** is that it enables clients to integrate easily into the ASAP model. This allows the Control Server to start each client in the same manner as it starts server applications.

To use this library, you must define the following functions:

- **appl_initialize()** – The application client initialization routine
- **appl_cleanup()** – The application client termination cleanup routine

Server application API library – libcontrol

The Server Application API library, **libcontrol**, provides you with routines specific to server applications. Each ASAP application server links to this library, so it provides a total environment from where you can develop server applications.

Using this library, the Control Server can start each server application and monitor its behavior while it is running. It adds considerable functionality to the server application before any user-specified functionality is added to the server.

This library provides you with considerable functionality including:

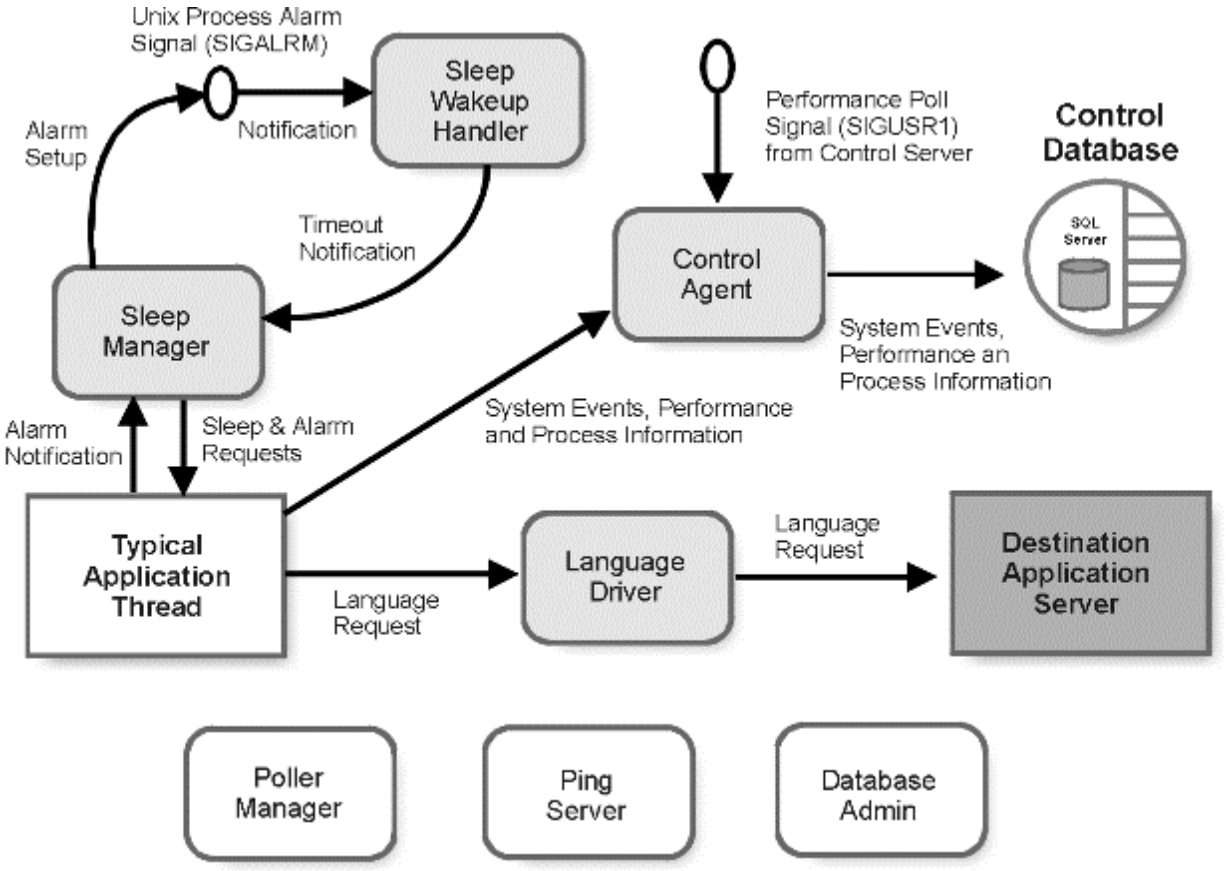
- Memory management
- Thread spawning and thread administration
- Performance monitoring and management
- Non-blocking thread versions of UNIX system calls: sleep(), alarm(), read(), write(), poll(), and so forth
- Control Agent routines controlling the Control database for performance parameter updates
- Database administration thread
- Pools of SQL server network connections to various databases for application use
- Language requests management
- Client connection and disconnection handlers
- Many administrative and diagnostic RPCs
- Library version registry

To use this library, you must define the following function:

- **appl_initialize()** – The application server initialization routine which can add RPC/ Registered Procedure/Language Request handlers, spawn application service threads, and so forth.

This server library also provides considerable functionality to any application server which links to this library. Figure 1-10 outlines a schematic of a server without any application functionality.

Figure 1-10 Server Schematic



The Server Application API library also contains an application server template to which you can add application-dependent functionality. This library also spawns a number of threads to manage some common functions.

The following subsections describe some of these service threads, as well as other functions that are not managed by service threads.

The server API discussed here is generally excluded from schematic descriptions of an application server in order to avoid confusion and to emphasize the functionality of the particular server. The functionality of the API is present in all ASAP application servers.

Sleep manager and sleep wakeup handler threads:

The Sleep Manager and Sleep Wakeup Handler threads run constantly in each application server.

The SYBASE Open Server library provides routines that let you put a thread to “sleep" on a particular object (**srv_sleep()**) and be “woken up" (**srv_wakeup()**) by another thread within the Open Server. Unlike UNIX, there is no function to put a thread to sleep for a specified time period. Therefore, to put a thread to sleep for a specified time period, you can use a separate API routine, **ASC_sleep()**.

The UNIX **alarm()** call is a shared resource. When activated, it issues a signal interrupt to the UNIX process, affecting all threads in a multithreaded process. In many

cases, a particular thread may wish to alarm only itself, and not impact the operation of other threads in the UNIX process.

For the above cases, the libcontrol library spawns two service threads when the application starts up (refer to the previous diagram). These threads are:

- **sleep manager thread** – This thread receives sleep and alarm requests from other threads within the process.
- **sleep wakeup handler thread** – This thread handles the UNIX **alarm()** call and passes the timeout notification to the sleep manager thread, which then forwards the alarm notification to the relevant application thread.

Control agent thread:

When the ASAP application server starts up, the Control library spawns the Control Agent thread, which does the following:

- Runs constantly in each application server.
- Opens a connection to the Control database in the SQL Server. Whenever an application thread issues a system event or database audit log or error entry, the thread calls a function in the API which, in turn, interfaces with the Control Agent thread to update the relevant database tables.
- Installs a UNIX signal handler for the SIGUSR1 signal. When this signal is received by the application server process in the Control Server, the Control Agent thread invokes functions in the Control database and updates that server's process and performance parameter information. The Control Server sends the performance poll signal based on the performance poll-interval period. The poll interval is configured from the Control Server.

Server language driver thread:

In some cases, one application server needs to send a large amount of textual information (NE response files, for example) over the network to another application server, possibly a remote machine.

To allow an application thread to send a text file or a text buffer as a Open Server language request to another server, use the API function **ASC_send_text()**. When this API function is called to send a text file to a particular application server for the first time, the API spawns a Server Language Driver thread. This thread continues running from that point onwards.

The Server Language Driver thread manages the network connection to the other application server and transmits the textual buffer to the destination application server. The destination application server triggers an Open Server language event.

To properly receive the language buffer, the destination server must have the appropriate language event handlers installed (using API functions). The server language driver thread maintains the network connection to that particular application server, so that any subsequent calls to this function to send textual information to this server are routed to this thread.

If the application calls the API function to send textual information to another application server, the API spawns a new server language driver to manage the new connection and data transmission. Therefore, you must set up one thread on the application server for every connection to a separate application server.

Database administration thread:

The Database Administration thread is a background thread that is present in every application server. It is spawned when the API server starts up. At a daily time that you

configure, this thread establishes a connection to the server's primary database and does the following:

- Performs database administration tasks (for instance, data archiving, purges, and reports) in a function and passes the procedure a parameter.

  You configure the function and the parameters, and specify the database administration tasks to perform.
- Updates statistics for all tables within the database to ensure the correct execution plan is chosen whenever the functions are recompiled.
- Recompiles all functions in the database.

When the above tasks are completed, the thread terminates the connection to the primary database.

Poller manager thread:

The Poller Manager thread manages API access to the Sybase **srv_poll()** routine to provide UNIX-like polling functionality to each application thread. It is invoked by means of the **ASC_poll()** API call.

Ping Server Thread:

The Ping Server thread is a constantly executing thread within each application server which periodically checks that both the application SQL and Control Server are running. If one or both are not running for any reason, then this thread will terminate the application server.

Interpreter API library – libinterpret

The Interpreter API library, **libinterpret**, provides routines for application servers that need to use State Tables. This library is generally used by SRPs, and NEPs.

Follow these steps to use this library for an application server:

1. Initialize the interpreter using the **ASC_init_interpreter()** API call.
2. Once initialized, use the **ASC_alloc_interpreter()** API call to allocate the library.
3. To deploy the library, use the **ASC_interpreter()** API call.
4. When the previous step is complete, free the library using the **ASC_free_intrepreter()** call.
5. If you want to add or overwrite State Table actions with customized action handler functions, use the **CMD_user_actions()** API call.

SRP API library – libsrp

The SRP API library, **libsrp**, shields each SRP from the details of SARM communication and notification responses. It provides a set of data structures and routines you can use to develop an SRP. The **libsrp** library has the following main components:

- Data structures and routines that describe the ASAP work order that the SRP translates into.
- Data structures and function pointers that process each notification returned to the SRP from the SARM.

- API routines that retrieve query information related to the processing on a particular ASAP work order from the SARM.

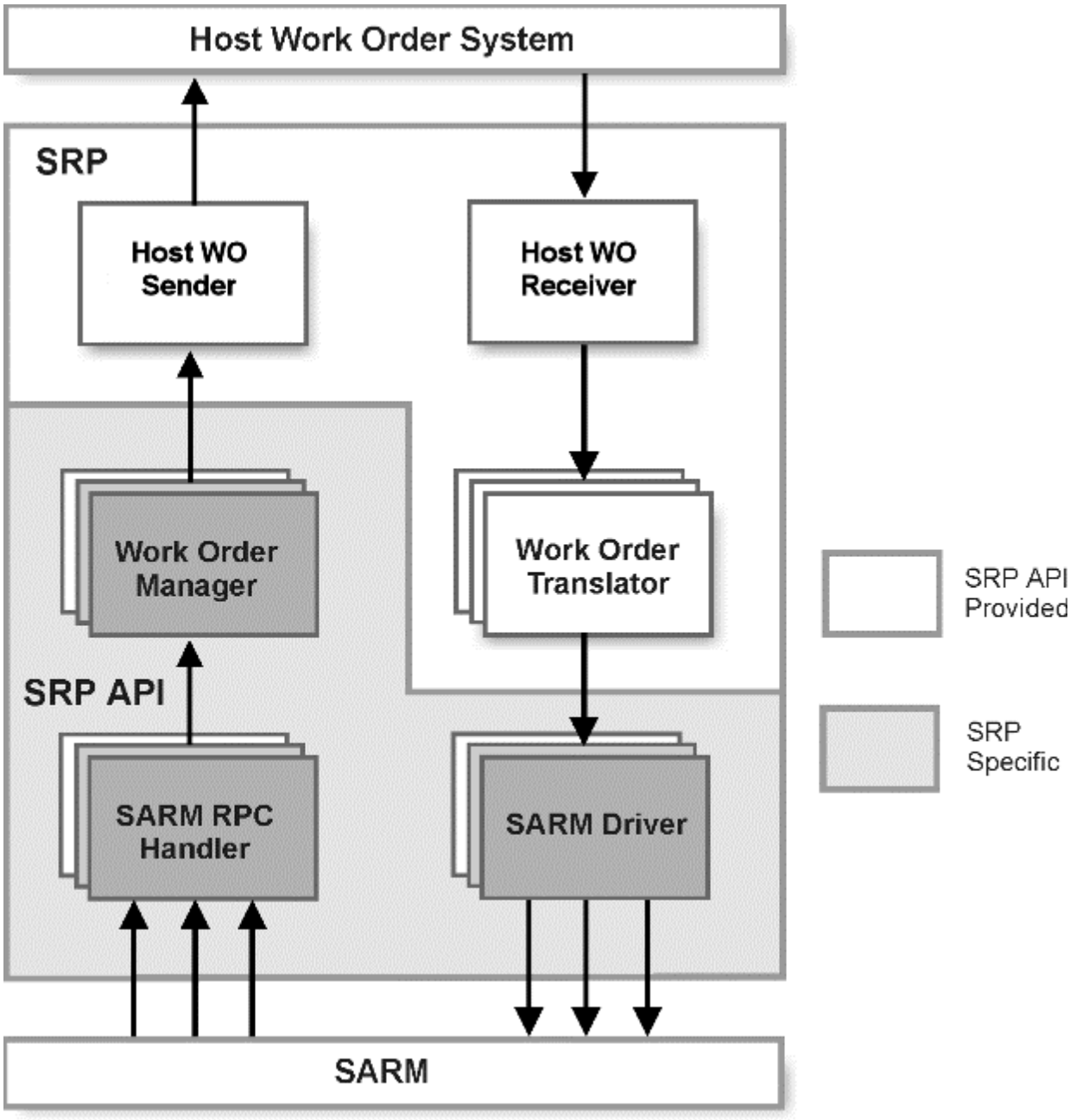
An SRP application links to the SRP API which contains both the Interpreter API library and the SRP API library. When the SRP initializes the SRP library using the **SRP_initialize()** API call, the API does the following:

- Initializes the Interpreter within the SRP so that the SRP can employ it (provided **libinterpret** has been compiled in the SRP application).
- Spawns SARM driver threads to manage the transmission of ASAP Work Orders to the SARM. The number of driver threads to be sent can be configured.
- Spawns work order manager threads to receive notification events returned from the SARM and calls user-specified functions to process each notification event it receives. The number of work order manager threads to be sent can be configured. The API also adds all registered procedures and RPCs that the SRP receives from the SARM.

> **Note:**
>
> ASAP tuning can increase ASAP performance, especially for work order management and handler threads.

Figure 1-11 shows a typical example of an SRP and SRP API components.

Figure 1-11 SRP and SRP API Components



NEP API library – libnep

The NEP API library, **libnep**, provides you with the tools to write an NEP. To use this library and generate a custom NEP server, you must provide the following API functions to the NEP “core" system:

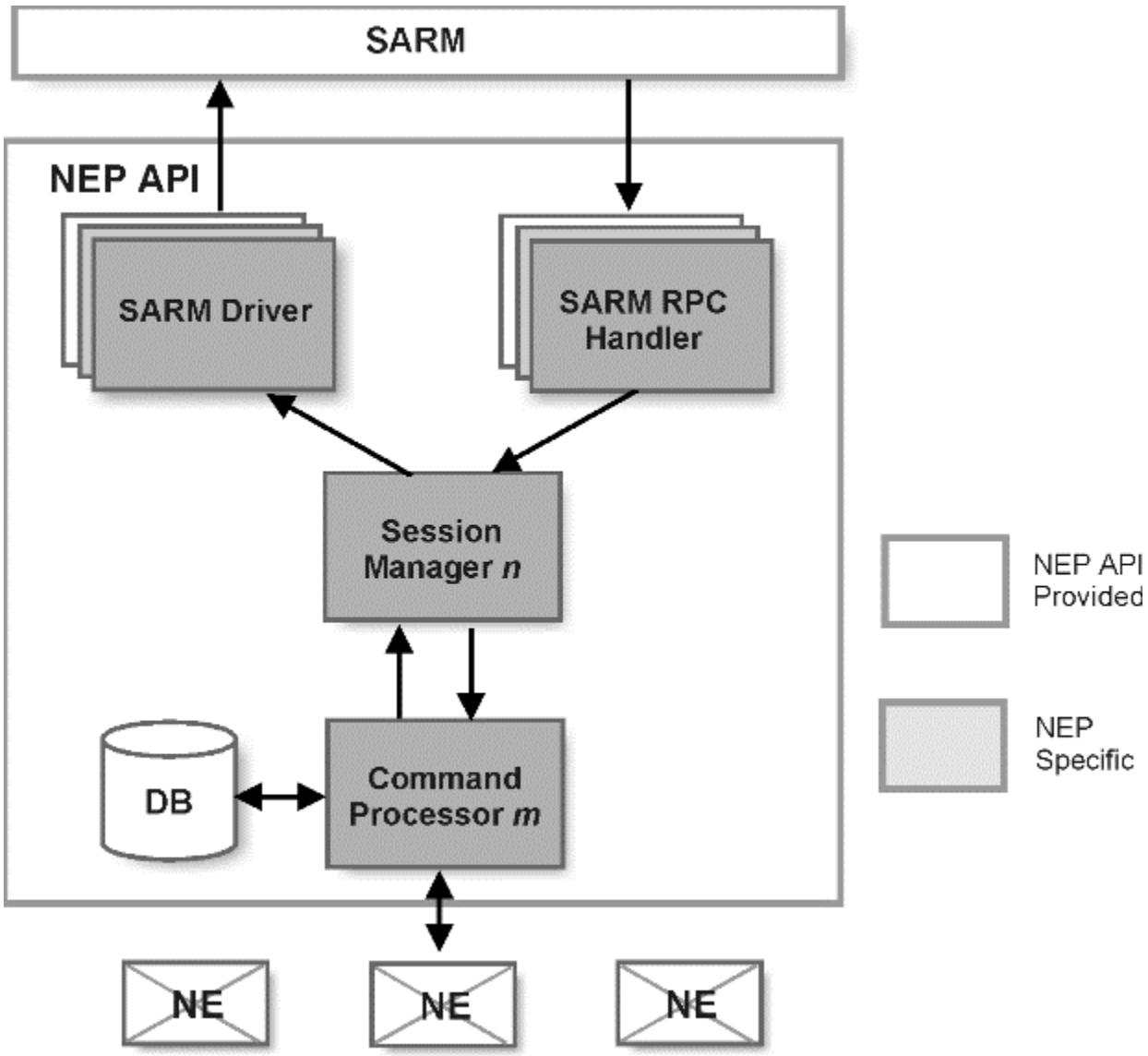
- **ASC_loadCommParams()** – Returns the list of communication parameters for the specified device type, host, and device.

  For more information, see "Interpreter library."

- **CMD_comm_init()** – Initializes the communications interface library. When the NEP requires interface-specific state table actions, use this function to register the actions using **CMD_user_actions** (see State Table Interpreter).
- **CMD_connect_port()** – Opens a connection to the device specified by the port information structure. This function also registers an association between the command processor that initiates the connect and the device.
- **CMD_disconnect_port()** – Closes the connection to the device specified by the port information structure

Figure 1-12 shows a typical example of an NEP and NEP API components.

Figure 1-12 NEP and NEP API Components



Multi-protocol communications API library – libasccomm

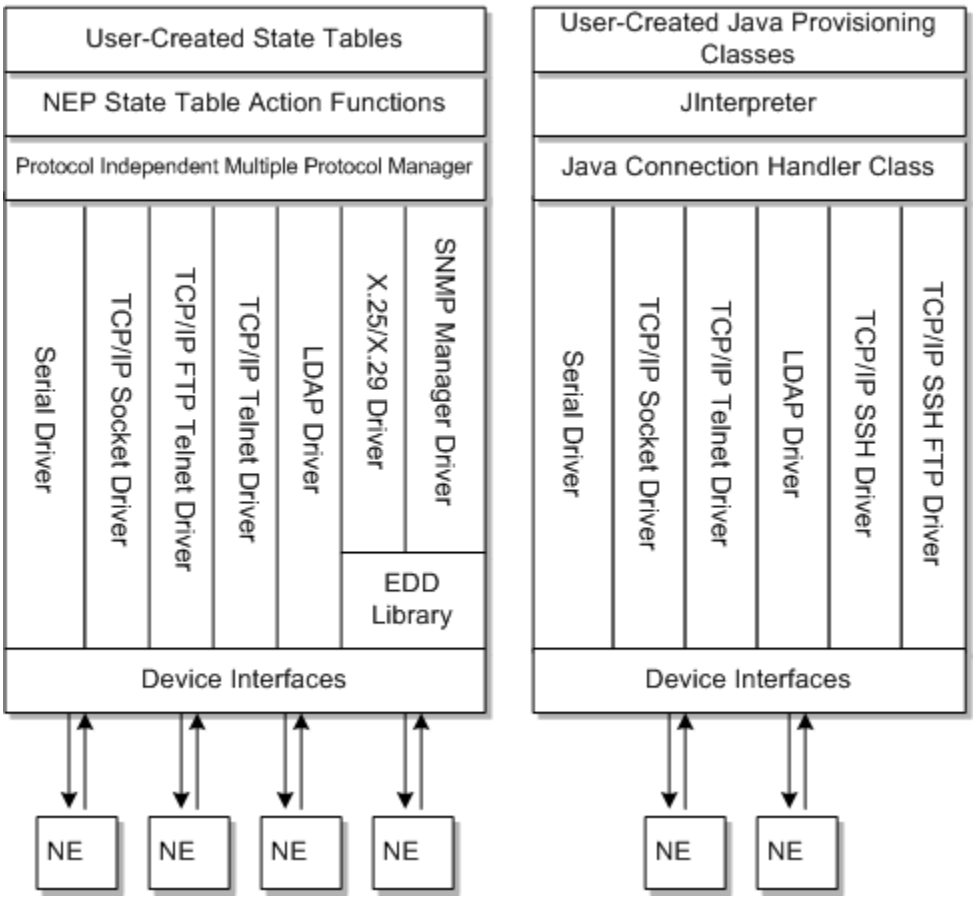
The communications library provides the Multi-Protocol Manager (MPM) with access to the various protocol drivers. Any ASAP Application Server that links in the interpreter library and the communications library, can communicate with a remote host using one of the supported device interfaces. Where terminal-based communication is required, the MPM routines manage the writing to and reading from the virtual screen.

The communications library allows one instance of an NEP application to communicate with hosts using multiple device interfaces. One command processor thread is dedicated to each remote host connection. To communicate with the device interface, the command processor invokes the MPM that is part of the communications API. The MPM will determine the appropriate device I/F handler based on the device interface type specified by the command processor.

For device interfaces that access UNIX devices and support non-blocking I/O, all the API functions are within the library. Hardwired or dialup (modem) serial interface, TCP socket, telnet, and SUN X.25 are some device interfaces that fall under this category. The generic driver will communicate with an external device driver using External Device Driver Interface API (**libgedd**) functions.

For device interfaces that need external device drivers due to a non-UNIX device interface API, or because nonblocking I/O is not possible, the communication library provides a generic driver to interface with the external device driver. For example, communicating to the host via the IBM-X25 API needs this approach because the IBM-X25 API cannot co-exist with the Open Server. The generic driver will communicate with an external device driver using the External Device Driver Interface API (**libgedd**) functions.

Figure 1-13 Logical NEP Command Processor Structure



Generic external device driver library – libgedd

The UNIX device API is managed from within the NEP using the Communications API. A non-UNIX device API requires an external device driver (EDD) to act as a gateway to transmit data between the NEP and the NE. The Generic External Device Driver API library, **libgedd**, handles the communication between the NEP and the EDD.

Network element configuration library – libnecfg

This library, **libnecfg**, provides commonly-used functionality that is NE-specific, MARCH-specific and blackout-related.

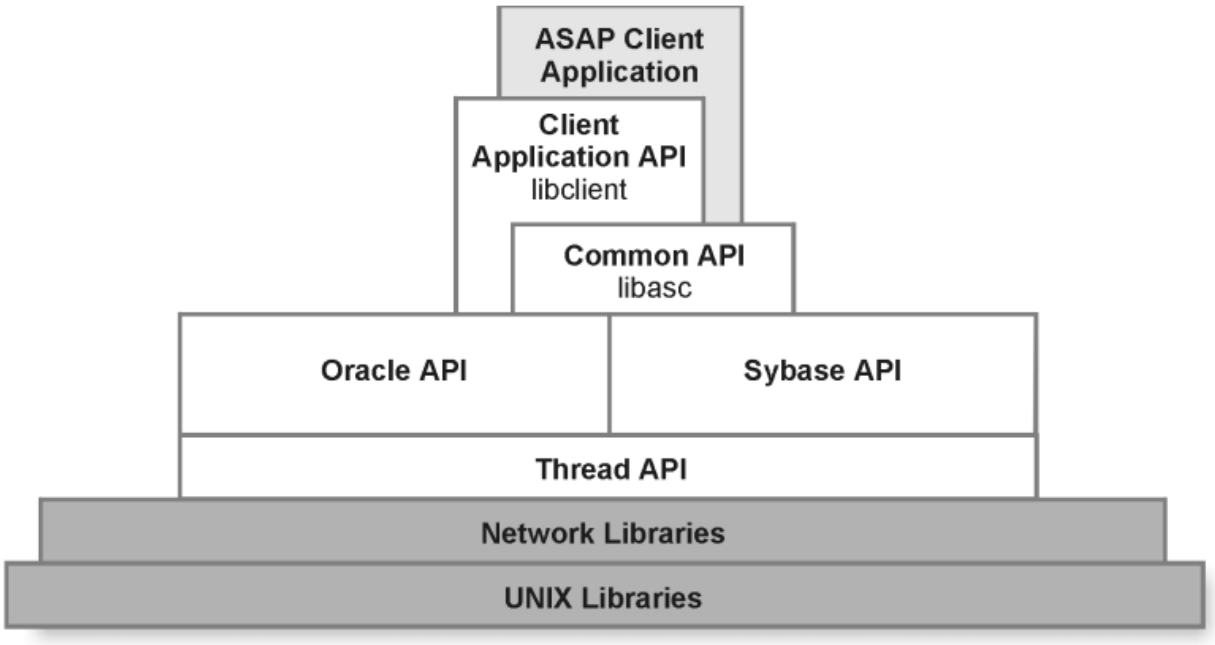
ASAP API application development

This section details the API structures used in various ASAP applications.

Client application structure

The Client application structure, as in Figure 1-14, is the API structure used within a typical ASAP client.

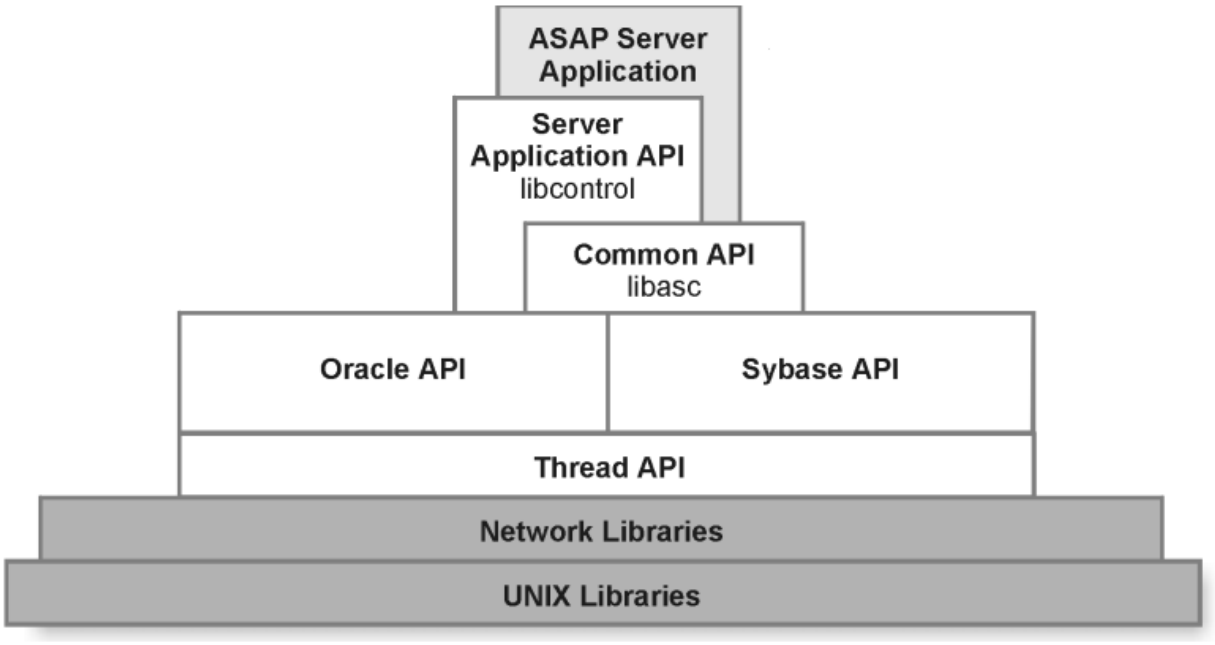
Figure 1-14 Client Application Structure



Server application structure

The Server application structure, as in Figure 1-15, is the API structure used within a typical ASAP server.

Figure 1-15 Server Application Structure

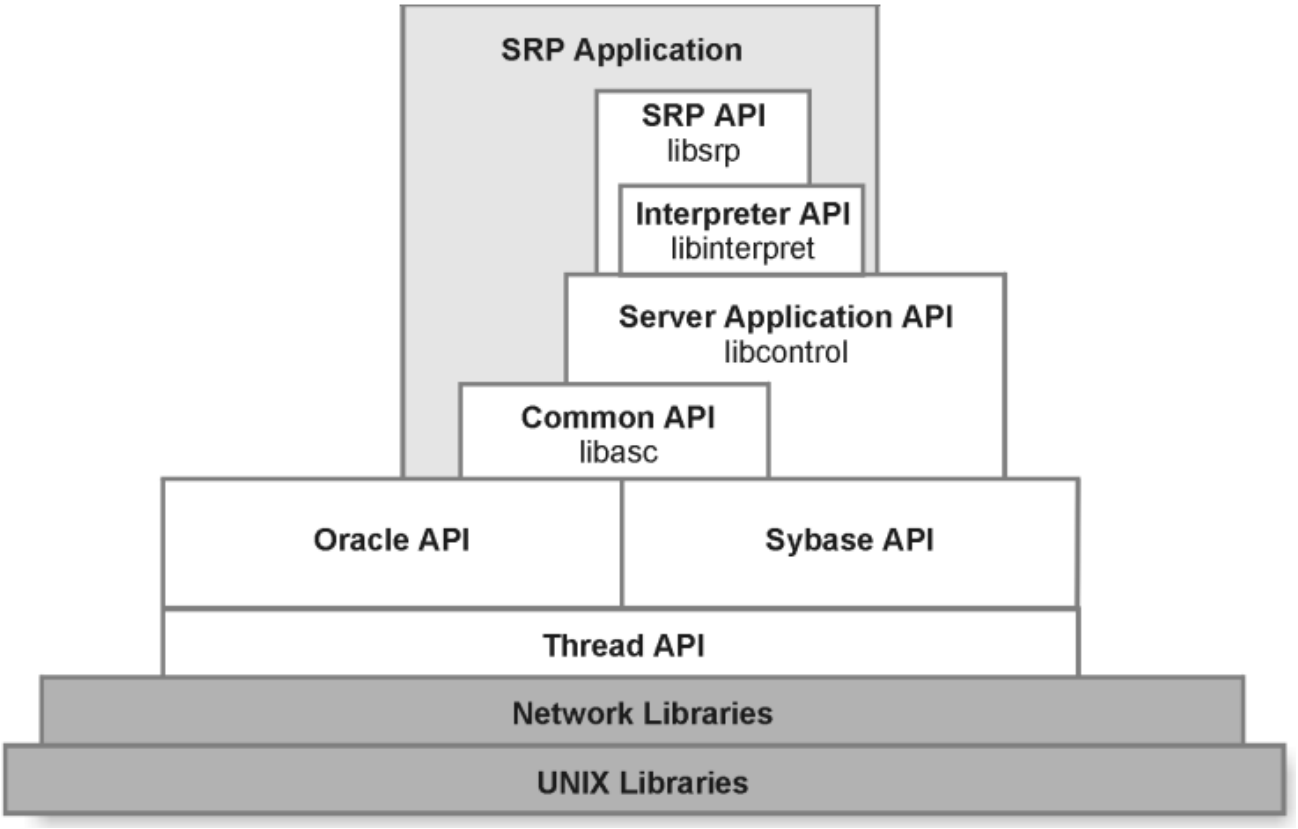


SRP server application structure

The SRP Server application structure, as in Figure 1-16, is the API structure used within a typical ASAP SRP server.

The SRP application is a server application, so it links in **libcontrol** and **libasc**, as well as **libsrp** and **libinterpret**.

Figure 1-16 SRP Server Application Structure

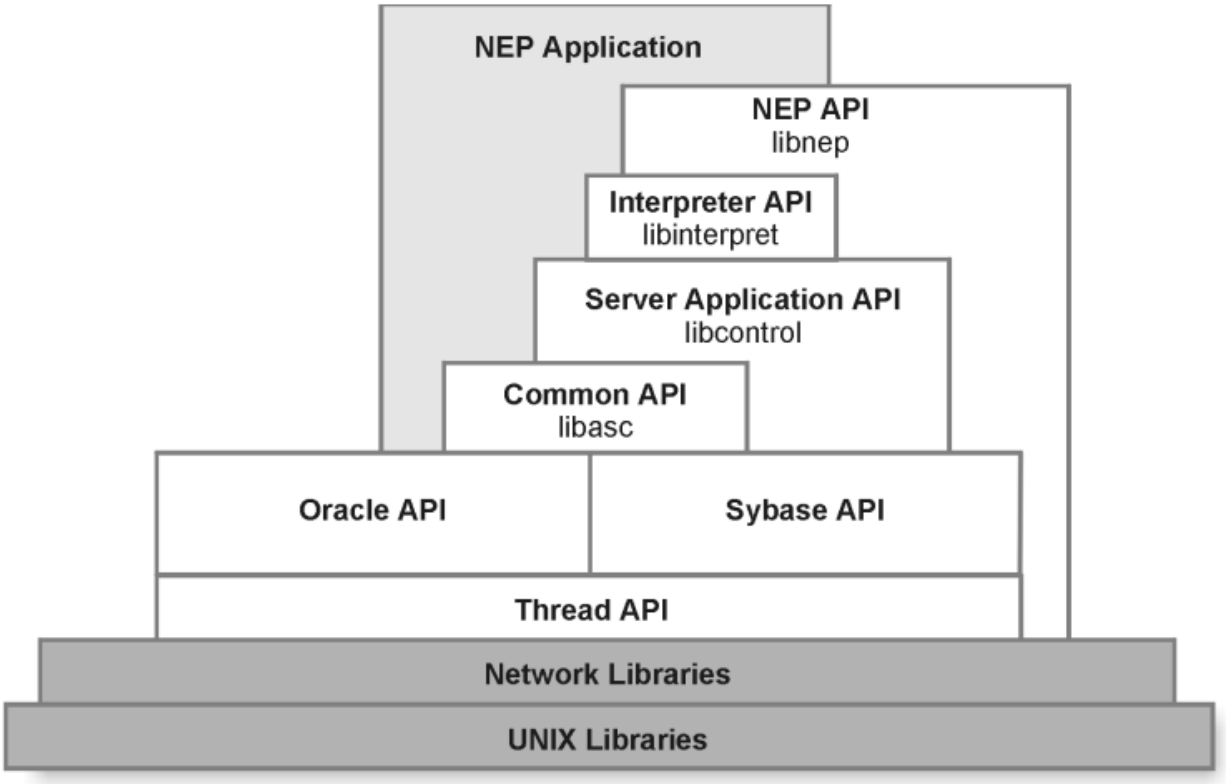


Generic NEP application structure

The Generic NEP application structure, as in Figure 1-17, is the API structure used within a typical Generic NEP server.

The NEP application is a server application, so it links in **libcontrol** and **libasc**, as well as in **libnep** and **libinterpret**.

Figure 1-17 Generic NEP Application Structure



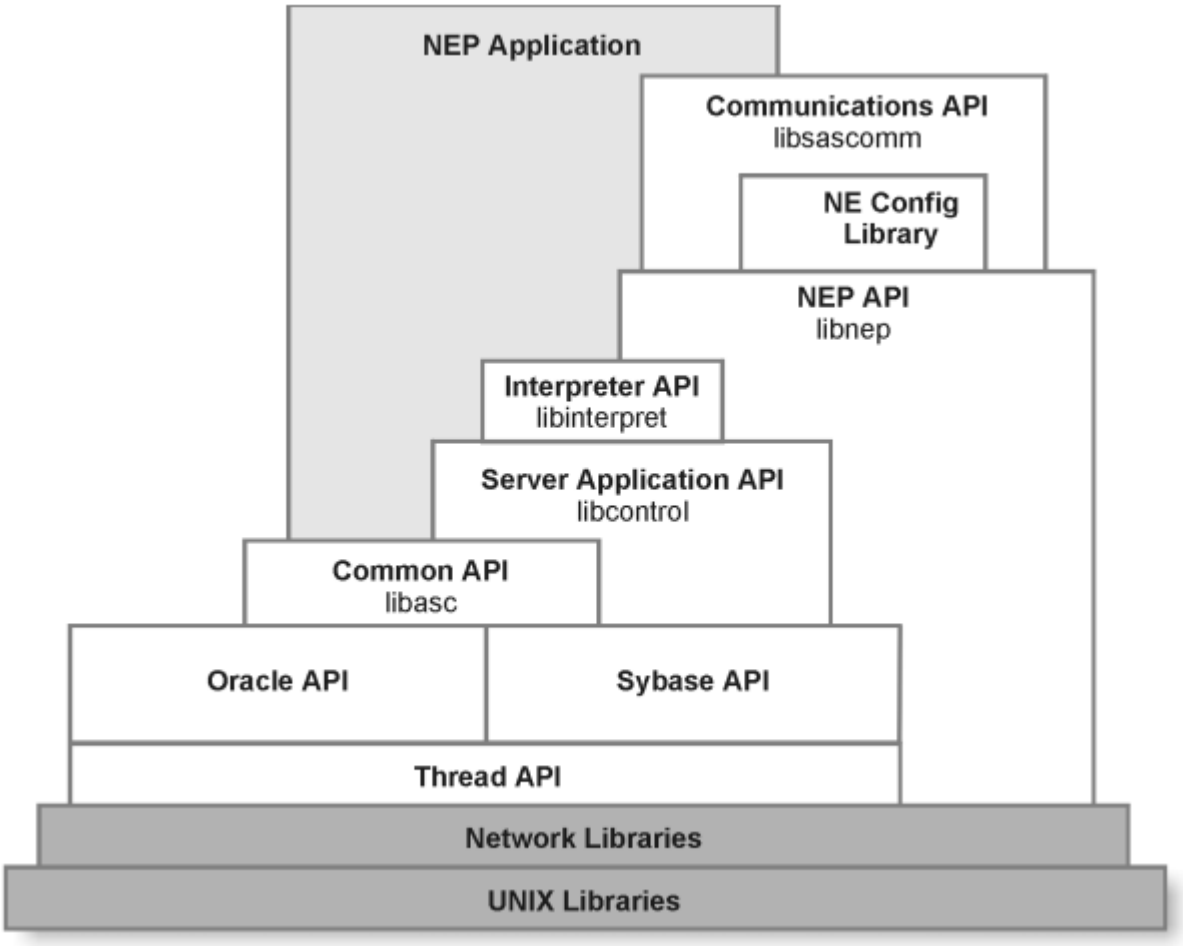
Multi-protocol NEP structure

The Multi-Protocol NEP structure, as in Figure 1-18, is the API structure used within a typical Multi-Protocol NEP server.

The NEP application is a server application, so it links in **libcontrol** and **libasc**, as well as in **libsrp** and **libinterpret**.

The Multi-protocol NEP application will link in libnep and libinterpret as is the case with the Generic NEP. In addition, it will link in the Communication library (**libasccomm**) and the Switch Configuration library (**libnecfg**).

Figure 1-18 Multi-protocol NEP Structure



Development of Cartridges supporting Asynchronous NEs

ASAP executes CSDLs and ASDLs in work orders synchronously. CSDLs and ASDLs are configured sequentially, and an ASDL must complete before the next ASDL can be started.

Some network elements respond to network actions asynchronously. After a request is sent, the *receipt* of the request may be acknowledged immediately but a response indicating *completion* of a request may be received from the network element some time later.

An ASDL in which the completion response arrives later is an asynchronous ASDL. A work order may have a mix of synchronous and asynchronous ASDLs mapped to the CSDLs.

Synchronous ASDLs require any previous ASDLs to be completed, so all asynchronous ASDLs run before a synchronous ASDL must complete before the synchronous ASDL starts.

> **Note:**
>
> Asynchronous Dynamic NEs are not supported.
>
> Asynchronous NE interfaces are supported through Java Enabled NEPs, and not through state table programming.

Asynchronous NE Response Handler

Instead of ASAP managing the processing of asynchronous ASDLs in the core, facilities are provided to the cartridge developer to handle them. These facilities include a response handler to manage asynchronous ASDL responses implemented in the java classes **ResponseHandler** and **ResponseHandlerManager**. Refer to the *ASAP Online Reference* for details on these classes.

When configuring asynchronous ASDLs, a 'stop work order' ASDL must follow the asynchronous ASDL(s) to halt work order processing. The NE response handler handles asynchronous responses from the NE, and resumes the work order when all outstanding asynchronous ASDL completion responses have been received.

Asynchronous connections:

Asynchronous NE interfaces have an entry in table **tbl_comm_param** with a parameter label **ASYNC_CONN** and parameter value of either **TRUE** or **FALSE**. See "tbl_comm_param."

When the NEP starts, for NEs that have an entry in table **tbl_comm_param** with a parameter label **ASYNC_CONN** and parameter value of TRUE, a connection is automatically established. If the parameter value is FALSE, the NE connection is not automatically established.

Each distinct asynchronous NE connection may have a distinct response handler. In this case, when an NE connection is released, then the associated response handler also should be stopped and removed from the system.

If multiple asynchronous NE connections use the same response handler, then the response handler may be spawned during the first NE connection and either left to keep running or stopped and removed from the JNEP by the cartridge developer when conditions as determined by the developer are met.

Response handler manager

Each Java-enabled NEP server (JNEP) uses a single instance of a response handler manager to manage all asynchronous response handlers within the JNEP. The response handler manager is accessed through static methods in its class. Typically, response handler creation is requested within the connect() method of the NE connection classes.

The response handler manager is implemented in the **ResponseHandlerManager** class. For details, refer to the *ASAP Online Reference.*

A sample implementation is provided with the ASAP product in

$ASAP_BASE/samples/JeNEP/async_ne

2
ASAP Database Tables

This chapter contains table information for the following databases:

- Control database
- SARM database
- NEP database
- Admin database
- C++ SRP API emulator database

Control database

This section describes the user-created database tables contained in the Control Database Data Dictionary.

User-created database tables

Following is the list of user-created database tables.

tbl_alarm_center

This static table defines the alarm centers where system alarm notifications are sent. You are responsible for maintaining this table. The initial data for this table is provided as part of the core system.

Table 2-1 tbl_alarm_center

Column_name	Type	Length	Nulls	Description
alarm_center	varchar2	8	0	The unique code representing the alarm center.
control_prog	varchar2	14	0	The program to be executed to communicate to the alarm to this alarm center. For example, this could be a shell script written to send a message over the network, to a printer or pager, etc., although any UNIX executable can be provided. The alarm program passes alarm data as command line arguments for interpretation by the alarm program. The core system provides a shell script and a sample C program, control_prog.c, which can be used as a basis to generate user-created alarm programs.
description	varchar2	80	1	A description of the alarm center.

Table 2-1 (Cont.) tbl_alarm_center

Column_name	Type	Length	Nulls	Description
opt#_type	varchar2	2	1	The option name passed to the control program, where # is a number between 1 and 5. This is a UNIX executable option to the control program. Can be null or any single wildcard character ("-_"), for example, -d -w, etc.
opt#_value	varchar2	20	1	The option value that is passed to the control program corresponding to the previous option type, where # is a number between 1 and 5.

Table indexes:

```
alarm_center
```

The following is an example of alarm centers **tbl_alarm_center**:

```
alarm_Center control_prog  description              opt1_type
 opt1_value      opt2_type opt2_value      opt3_type
 opt3_value      opt4_type opt4_value      opt5_type
 opt5_value
 ------------ -------------- ---------------------------------------- ---------

 ADMIN     admin.sh    Administration Alarm           NULL
           NULL          NULL    NULL                 NULL
           NULL          NULL    NULL                 NULL
           NULL
 ADMINPGR   adminp.sh   Administration Pager             NULL
            NULL          NULL    NULL                   NULL
            NULL          NULL    NULL                   NULL
            NULL
```

Each alarm center can invoke a program. This program is a UNIX executable or shell script. If set to **NULL**, no alarm program is invoked. In this example, two alarm programs, **admin.sh** and **adminp.sh**, are defined.

tbl_alarm_log

This dynamic table contains all of the system generated alarms.

Table 2-2 tbl_alarm_log

Column_name	Type	Length	Nulls	Description
event_unid	number	38	0	The unique ID of the event generating the alarm. If an alarm is generated by this event, an alarm log entry is created in tbl_alarm_log with this event_unid value.
alarm_code	varchar2	8	0	The alarm code of the generated alarm.
alarm_unid	number	38	0	The unique ID that identifies the alarm.
start_dts	date	-	0	The start date and time of the system alarm.

Table 2-2 (Cont.) tbl_alarm_log

Column_name	Type	Length	Nulls	Description
escalation_dts	date	-	1	If set, the date and time of the last alarm escalation.
clear_dts	date	-	1	If set, the date and time at which the alarm was cleared.
ack_user	varchar2	30	1	The user ID of the user who acknowledged the alarm.

Table indexes:

```
Indexes

alarm_unid
event_unid
alarm_code
```

tbl_appl_proc

This static table contains ASAP application configuration information. The ASAP startup procedure uses this table to determine the applications to start, and their start sequence.

You are responsible for maintaining this table. The initial data for this table is provided as part of the core system.

Table 2-3 tbl_appl_proc

Column_name	Type	Length	Nulls	Description
start_seq	number	38	0	Controls the sequence in which the applications are started. For example, certain client applications may be required to start before server applications, and other client applications after the server applications.
appl_type	char	1	0	Specifies the ASAP application type. • S – server • C – client • M – master control server • R – remote control server
appl_cd	varchar2	8	0	The logical ASAP application code, for example, SARM, NEP01, NEP02, etc. The ASAP configuration file contains entries for the real name of each application. The real name appears as a parameter value for the application's logical name.

Table 2-3 (Cont.) tbl_appl_proc

Column_name	Type	Length	Nulls	Description
control_svr	varchar2	8	0	The name of the logical ASAP application Control server that spawns the application and monitors its behavior. In a distributed ASAP configuration, there must be a Control server defined on each machine. In each ASAP configuration there is only one master Control server. This server is defined as having itself as a Control server. Remote Control servers are defined as having the master Control server as their Control server.
description	varchar2	80	1	A description of the ASAP application.
diag_file	varchar2	14	0	The name of the diagnostics log file where diagnostic messages are written. This file is created in the $LOGDIR directory under a dated directory, for example, $LOGDIR/yymmdd.
auto_start	char	1	0	An auto start flag for the application. Values: • Y – yes • N – no When starting ASAP with the command start_asap_sys, all applications with the autostart value of Y are started automatically. To configure the SRP in tbl_appl_proc, you must set auto_start to 'N'.
program	varchar2	40	0	The name of the UNIX executable to run. This UNIX program must reside in the $PROGRAMS directory and must be executable.
diag_level	varchar2	4	0	The diagnostic level of the ASAP application. The diagnostic level determines whether to log diagnostic information when the ASC_diag() API function call is used. Possible values are: • PROG • SANE • LOW • KERN
isactive	char	1	0	Flag denoting whether the ASAP application is currently active. This stops attempts at starting more than one instance of the same application. Set by CSP_start_server/CSP_start_client and reset by CSP_halt_server/CSP_halt_client functions. Possible values are: • N – ASAP application is not running. • Y – ASAP application is currently running.

Table 2-3 (Cont.) tbl_appl_proc

Column_name	Type	Length	Nulls	Description
last_start	date	-	1	If set, identifies the last start date and time of the ASAP application. It is set by the administrative Control function, CSP_start_server.
last_halt	date	-	1	If set, identifies the last halt or termination date and time of the ASAP application. It is set by the administrative Control function CSP_halt_server/CSP_halt_client.
last_abnormal	date	-	1	If set, identifies the last abnormal termination of the ASAP application process. This is determined by the ASAP application Control server.
svr_type	varchar2	8	1	Possible values include: • ADM – Admin server • CTRL – Control server • MASTER – Master Control server (must be only one/system) • SARM – SARM server • SRP – SRP server • NEP – NEP server

Table indexes:

```
Indexes

appl_cd
start_seq
```

tbl_classA_secu

tbl_classA_secu is an internal ASAP table that contains login ID and password information for each ASAP server.

ASAP class A secure data includes the database login/password for each ASAP server. This information is initially configured during ASAP installation.

The ASAP security administrator can maintain secure data by using the ASAP security tool. The ASAP security tool enables users to initialize and maintain security for ASAP Class A components (each core ASAP server). Refer to the *ASAP System Administrator's Guide* for more information.

In addition to the ASAP security administator, the following action functions enable users to access the secure data: GET_SECUREDATA, SET_SECUREDATA. For more information on these action functions, see "GET_SECUREDATA" and "SET_SECUREDATA" respectively.

Table 2-4 tbl_classA_secu

Column_name	Type	Length	Nulls	Description
name	varchar2	80	0	Name of key entry.

Table 2-4 (Cont.) tbl_classA_secu

Column_name	Type	Length	Nulls	Description
value	varchar2	255	0	Encrypted value.
class	integer	-	0	Possible values: • 0 – ASAP class A secure data.
secu_level	integer	-	0	One of the following: • 0 – Security feature is disabled. Password information appears in plain text • 1 – Security feature is enabled and passwords are encrypted.
s_cache	integer	-	0	Reserved for future use.
audit_level	integer	-	Reserved	Reserved for future use.
c_date	date	-	0	Creation date.
alg	integer	-	0	Type of cypher algorithm, currently supports only BLOWFISH alg (1).
desc1	varchar2	255	1	Description.

Table indexes:

```
name, value
```

tbl_classB_secu

tbl_classB_secu contains user ID, password, and other information used for custom components, particularly the C SRP API, C++ SRP API, Java-enabled NEP, and NEs.

User-defined secure data can be set up in the following ways: using the ASAP security administration tool or through APIs or action functions.

To import a large amount user-defined secure data into ASAP secure storage, it is recommended that the user compose a flat file containing essential secure data information. The format is "name:value:description" with semicolons used as delimiters. For example, a data file will reference secure data as follows:

```
TOR_NE:password1:Class B NE login info
ENG_NE:password2:Class B NE login info
```

> **Note:**
>
> The ASAP security administration tool encrypts the value field.

Refer to the *ASAP System Administrator's Guide* for more information.

The user can access the secure ASAP data through security APIs or action functions.

The following security APIs from customized SRP and Java-enabled NEP. These security APIs include ASC_get_securedata, ASC_set_securedata. For more information on these APIs, see "ASC_get_securedata" and "ASC_set_securedata" respectively.

The following action functions enable users to access the secure data from customized or non-customized NEPs: GET_SECUREDATA, SET_SECUREDATA. For more information on these action functions,see "GET_SECUREDATA" and "SET_SECUREDATA" respectively.

Table 2-5 tbl_classB_secu

Column_name	Type	Length	Nulls	Description
name	varchar2	80	0	Name of key entry.
value	varchar2	255	0	The encrypted value.
class	integer	-	0	Reserved for future use.
s_cache	integer	-	0	Reserved for future use.
c_date	date	-	0	Creation date.
desc1	varchar2	255	1	Description.

Table indexes:

```
name, value
```

tbl_code_list

This static table is maintained within the Control database. It is used to track core and custom code used in ASAP. For instance, this table can identify code that tracks cartridges deployed within ASAP.

Table 2-6 tbl_code_list

Column_name	Type	Length	Nulls	Description
code_type	varchar2	4	0	Type of code.
code	varchar2	80	0	The code entry.
value	varchar2	4	0	The code value.
code_desc	varchar2	25	1	A description of the code.
parm1	varchar2	25	1	General purpose parameter field.
parm2	varchar2	25	1	Same as parm1.
parm3	varchar2	25	1	Same as parm1.
parm4	varchar2	25	1	Same as parm1.

Table indexes:

```
code_type, code, value
```

tbl_component

This static table contains a list of ASAP processes for each ASAP territory and system. You are responsible for populating and maintaining this table.

Table 2-7 tbl_component

Column_name	Type	Length	Nulls	Description
territory	varchar2	20	0	The name of the ASAP territory. This allows the components of all ASAP systems in all territories to be specified and maintained in a single database. It is possible to have many territories in an ASAP implementation. Territories are generally mutually exclusive since there is no communication between ASAP systems in different territories.
system	varchar2	20	0	The name of the ASAP system within an ASAP territory.
component	varchar2	40	0	The ASAP application component within this territory. This refers to the application components listed in the appl_cd column in tbl_appl_proc.

Table indexes:

```
territory, system, component
```

tbl_db_threshold

You can configure this static table to enable monitoring of data and transaction log sizes for a specific database by the ASAP Control server. You can specify the thresholds and the system events to be generated if such thresholds are exceeded. The Control server periodically performs this check.

> **Note:**
>
> The interval between checks is configured by the Control server application configuration variable, DB_MONITOR_TIME.

For information on DB_MONITOR_TIME, refer to the *ASAP System Administrator's Guide*.

The following is a an example of a database monitoring configuration.

Table 2-8 Sample database monitoring configuration

asap_sys	db_name	data threshold (%)	data event
PROD	POOL_TS	85	MAJOR-EVT
PROD	POOL_TS	95	MAJOR-EVT

> **Note:**
>
> For the db_name column, you must use the specific name of the tablespace (not the database name) where the databases that you want to monitor are located.

In this database configuration example, the MAJOREVT system event is generated on the production system if the POOL_TS database becomes 85% full. If the transaction log for the database exceeds 100 MB, the CRIT_EVT is generated by the system.

Table 2-9 tbl_db_threshold

Column_name	Type	Length	Nulls	Description
asap_sys	varchar2	8	0	The ASAP environment. This should be set equal to the value of the appropriate ASAP_SYS environment variable. The Control server only reads records from this table with values of asap_sys equal to the currently defined ASAP_SYS environment variable.
db_name	varchar2	80	0	The name of the database in this asap_sys environment, for example, SDB_U01_asap.
data_threshold	number	38	0	The data threshold above which the system event data_event is issued (in percentage 0-100).
tran_threshold	number	38	1	Obsolete.
data_event	varchar2	8	0	The system event to be issued if the data_threshold is exceeded.
tran_event	varchar2	8	1	Obsolete.

Table indexes:

```
asap_sys, db_name
```

tbl_event_log

This dynamic table contains a log of all system events generated by ASAP applications.

You can create these entries by using the ASC_event() API function from within the source code.

Table 2-10 tbl_event_log

Column_name	Type	Length	Nulls	Description
appl_cd	varchar2	8	0	The logical name of the ASAP application that generated the system event.
event_type	varchar2	8	0	The event type that specifies whether a system alarm is to be generated by accessing the static table, tbl_event_type.

Table 2-10 (Cont.) tbl_event_log

Column_name	Type	Length	Nulls	Description
event_unid	number	38	0	A unique ID of this event. If an alarm is generated by this event, an alarm log entry is created in tbl_alarm_log with this event_unid value.
source_file	varchar2	14	0	The source file name where the event was generated.
source_line	number	38	0	The line in the source file where the event was generated.
reason	varchar2	80	0	A description of the system event.
event_dts	date	-	0	The date and time of the system event.
ack_user	varchar2	30	1	ID of the user who acknowledged the event.

Table indexes:

```
Indexes

event_unid
appl_cd, event_dts
```

tbl_event_type

This static table defines the system events an ASAP application may generate and if required, the system alarm code associated with the event. You can generate system events using customer-specific code. You can configure these system events to enable or disable system alarms, as required.

System events are logged in the Control database table, tbl_event_log. The ASAP core system specifies the system events generated by the core system. You are responsible for updating the existing system events and adding your own specific events.

This table is referenced by the customEvent element in ServiceModel.xsd. In other words, The customEvent must reference an event that has been defined in tbl_event_type in the control database.

Table 2-11 tbl_event_type

Column_name	Type	Length	Nulls	Description
event_type	varchar2	8	0	The ASAP event type generated by the application. Each system event must have a record in this table. The core API includes the following system events: • ABNORMAL – Abnormal Process Termination - Application Terminated Unexpectedly. • SYS_TERM – Application Self Termination upon Critical Error. • SYS_ERR – General Application Process Error. • SYS_WARN – General Application Process Warning. • SYS_INFO – General Application Information. • DISK_ERR – Critical Disk / Media Error. • RPCSPACE – Critical Database Resource Error. • RPC_ERR – General Application RPC Error. • UNIX_ERR – UNIX System Call Error. • NETWK_ERR – Application Network Connection Error. • SRVOBJER – Application Server Object Access Error.
				The Control server can issue the following events: • APP_STRT – ASAP Application Local or Remote Startup. • APP_STOP – ASAP Application Local or Remote Shutdown. • APP_ERR – ASAP Application Startup Error.

Table 2-11 (Cont.) tbl_event_type

Column_name	Type	Length	Nulls	Description
event_type (continued)	varchar2	8	0	The SARM can issue the following events: • WOINPROC – Warning: Work Order(s) in Progress Longer than Specified Threshold. • ROUT_ERR – Error: SARM Routing Error - Unable to Determine Host NE. The NEP can issue the following events: • MAINTNCE – Information: Host NE has gone into Maintenance Mode. • BIND_ERR – Warning: Unable to Allocate Device to Connect to NE. • CONN_ERR – Warning: NE Connection Attempt Failed. • DIAL_ERR – Warning: Dial-up Program to Connect to NE Failed. • SYS_TUNE – Informational message that ASAP is auto-tuning its connection pools. • LOGN_ERR – Warning: Login Program to Network Element Failed. • PORT_DIS – Error: Connection to NE Failed; Port/Device DISABLED.
description	varchar2	80	1	A description of the event. If an alarm is generated by this event, the description is passed to the alarm program.
alarm_code	number	8	1	The alarm code associated with the system event. If null, a database log entry is created, but no alarm is generated.
alarm_action	varchar2	2	1	The alarm action; specifically, the enabling or disabling of the associated system alarm. It allows system events to generate and stop system alarms. Possible values are: • E – Enable an alarm. • D – Disable an alarm.
notify_aims	char	0	1	Reserved for future use.

Table indexes:

```
event_type
```

The following is an example of the tbl_event_type table:

```
event_type description                                    alarm_code
alarm_action
 ---------- ---------------------------------------- ---------- ------------
ABNORMAL  Abnormal Process Termination - Application     ABNORMAL   E
DB2FULL   Database Space is Critical                     CRIT_NAC   E
DISK_ERR  Critical Disk / Media Error - See Diagno       CRIT_AC    E
FS2FULL   File System Space is Critical                  CRIT_NAC   E
MAINTNCE  Information: Host NE has gone into Maint       MIN_AC     E
NTWK_ERR  Application Network Connection Error - S       MIN_AC     E
ROUT_ERR  Error: SARM Routing Error - Unable to De       MIN_AC     E
RPCSPACE  Critical Database Resource Error - See D       CRIT_NAC   E
RPC_ERR   General Application RPC Error - See Diag       MIN_AC     E
```

```
SRVOBJER  Application Server Object Access Error -       MIN_AC     E
SYS_ERR   General Application Process Error              MAJ_AC     E
SYS_INFO  General Application Information
SYS_WARN  General Application Process Warning                       E
TRANFULL  Tranlog Space is Critical                      CRIT_NAC   E
UNIX_ERR  UNIX System Call Error - See Diagnostics       MIN_AC     E
WOINPROC  Work Order(s) in Progress Longer than Sp       MIN_AC     E
```

tbl_fs_threshold

You can configure this static table to enable monitoring of the desired UNIX file system sizes by the ASAP Control server. If the thresholds are exceeded, you may specify both the thresholds and the system events to be generated. The Control server periodically performs this check.

> **Note:**
>
> You can configure the interval between checks by using the Control server application configuration variable, FS_MONITOR_TIME.

A possible configuration for this table is:

Table 2-12 Sample FS_MONITOR_TIME table

asap_sys	file_system	full threshold (%)	full event
PROD	/	80	MAJOREVT
PROD	/DATA	85	MAJOREVT

With this threshold configuration, the MAJOREVT system event is generated on the test system if the “/" file system becomes 80% full. This function is platform-specific and may not work on all platforms.

The implementation of the file system monitor varies across different platforms. Before defining the FS threshold, consult your UNIX Administrator for the file system configuration.

ASAP reports less than what the UNIX file system actually uses since ASAP takes into account the 10% minimum free space requirement when determining the file system space available. Therefore, you must set the FS threshold lower than the actual threshold level. For example, if the file system full alarm FS_FULL is to be generated at an 80% level, you should set the FS threshold at 75%.

Table 2-13 tbl_fs_threshold

Column_name	Type	Length	Nulls	Description
asap_sys	varchar2	8	0	The ASAP environment. The Control server only reads records from the table with values of asap_sys equal to the currently defined ASAP_SYS environment variable.
file_system	varchar2	80	0	The name of the file system.

Table 2-13 (Cont.) tbl_fs_threshold

Column_name	Type	Length	Nulls	Description
full_threshold	number	38	0	The threshold above which the system event full_event is issued (in percentage 0-100).
full_event	varchar2	8	0	The system event issued should the full_threshold be exceeded.

Table indexes:

```
asap_sys, file_system
```

tbl_listeners

This table allows any ASAP server to set up socket listeners for receiving RPC requests. You must configure this table to allow the SARM to start up socket listeners for incoming SRP requests. As well, every Java-enabled NEP must maintain a dedicated connection to its JInterpreter.

Table 2-14 tbl_listeners

Column_name	Type	Length	Nulls	Description
srv_name	varchar2	8	0	Name of the server that starts a socket listener. The SARM must start a socket listener to receive incoming Java SRP requests. For a Java-enabled NEP, this is the name of the NEP ($NEP). For the Java SRP, this column contains the SARM name.
host_name	varchar2	80	0	The host name or the IP address on which the server application resides. For the JInterpreter, this value must always be localhost. For the Java SRP, the host_name identifies the location of the SARM.
listener_name	varchar2	40	0	The name of the listener thread. For a Java-enabled NEP, the listener name describes the listener in the Java process that accepts interpreter requests from the C process. This listener name must always be $NEP_jlistener. For the Java SRP, observe the naming convention of "<Java SRP application name>_jsrplistener". This column is used by the Java SRP to retrieve the listener configurations.
port	integer		0	A free port on which the server can start the socket listener.

Table indexes:

```
srv_name, listener_name
```

tbl_name_value_pair

This static table provides the facility to maintain miscellaneous name value pair information related to the Control database. You are responsible for maintaining this table.

Table 2-15 tbl_name_value_pair

Column_name	Type	Length	Nulls	Nulls
name	varchar2	40	0	The name of the parameter.
value	number	38	0	The value of the parameter.

Table indexes:

```
name
```

tbl_process_info

This table contains information on process system resource usage. This information is updated according to the frequency of polling, as specified by the PERF_POLL_PERIOD configuration variable.

Table 2-16 tbl_process_info

Column_name	Type	Length	Nulls	Nulls
appl_cd	varchar2	8	0	The name of the ASAP server.
info_dts	date	-	0	The date and time of the ASAP process performance polling.
sys_events	smallint	-	0	The total number of system events generated by appl_cd since ASAP start up.
user_cpu	integer	-	0	User CPU time.
system_cpu	integer	-	0	System CPU time.
proc_identity	number	20	0	The ID of the process.

Table indexes:

```
appl_cd, info_dts
```

> **Note:**
>
> The user_cpu and system_cpu are gathered by each ASAP server by using the times() UNIX system call. The times are in units of 1/CLK_TCK seconds.

tbl_server_info

This static table stores OCA SRP server information required for the OCA applet to connect to different sessions. When the OCA SRP server starts, it sends an HTTP request to the IORManager servlet, which extracts the following information from the HTTP request:

- OCA SRP server name
- Name of host where WebLogic Server is deployed
- Port number on which WebLogic Server is listening

The servlet then inserts the server, host name and port number string in the table.

When the OCA applet starts up, it reads the following information from the table.

Table 2-17 tbl_server_info

Column_name	Type	Length	Nulls	Description
servername	varchar2	8	0	The name of the server.
hostname	varchar2	80	0	The host on which the WebLogic server resides.
info	varchar2	512	0	The port number on which WebLogic Server is listening.

For more information on configuring the OCA, refer to the *ASAP Installation Guide*.

tbl_system_alarm

This static table describes ASAP system alarms that can be generated by ASAP system events. You must populate and maintain this table. Some initial data is provided in the core system.

Table 2-18 tbl_system_alarm

Column_name	Type	Length	Nulls	Description
alarm_code	varchar2	8	0	The alarm code.
description	varchar2	80	0	A description of the system alarm that is passed to the alarm program.
alarm_level	varchar2	8	0	The level of the alarm. Possible values are: • MINOR – minor alarm. • MAJOR – major alarm. • CRITICAL – critical alarm.
escalation_code	varchar2	8	0	Reserved for future use.
escalation_time	number	38	0	Reserved for future use.

Table 2-18 (Cont.) tbl_system_alarm

Column_name	Type	Length	Nulls	Description
auto_clear	char	1		If this flag is set to Y, it automatically clears the alarm after being generated; otherwise, the alarm continues to go off until it is either manually turned off or a system event is triggered to disable it. For more information, see "tbl_event_type." You can use a Control Server administrative RPC, alarm_stop, to turn off a particular non-auto clearing system alarm.
route#_period	number	38		Where # is a number between 1 - 5. The period in minutes from the alarm to the alarm center.
route#_start	number	38		Where # is a number between 1 - 5. The daily start time in minutes after midnight for the alarms to go to the alarm center. A value between 0 and 1440.
route#_end	number	38		Where # is a number between 1 - 5. The daily end time in minutes after midnight for the alarms to go to the alarm center. A value between 0 and 1440.
route#_center	varchar2	8		Where # is a number between 1 - 5. The alarm center to route alarms.

Table indexes:

```
alarm_code
alarm_code description               alarm_level escalation_code

escalation_time auto_clear route1_period route1_start route1_end
route1_Centre route2_period route2_start route2_end route2_Centre

---------- ---------------------------------------- ----------- ---------------
ABNORMAL  Abnormal Process Termination       CRITICAL

 NULL N           5      0    1440
 ADMINPGR        NULL     NULL    NULL

CRIT_NAC  Critical ASAP System Alarm        CRITICAL

 NULL N           5      0    1440
 ADMINPGR        NULL     NULL    NULL

CRIT_AC  Critical ASAP System Alarm        CRITICAL

      NULL Y           5      0    1440
 ADMINPGR        NULL     NULL    NULL

MAJ_NAC  Major ASAP System Alarm         MAJOR

      NULL N           5      0    1440
 ADMIN         NULL     NULL    NULL

MAJ_AC   Major ASAP System Alarm         MAJOR
```

```
      NULL Y           5       0    1440
 ADMIN          NULL      NULL    NULL

MIN_NAC  Minor ASAP System Alarm          MINOR

      NULL N           5       0    1440
 ADMIN          NULL      NULL    NULL

MIN_AC   Minor ASAP System Alarm          MINOR

      NULL Y           5       0    1440
 ADMIN          NULL      NULL    NULL
```

tbl_unid

You can use this dynamic table to manage unique IDs needed by other tables. It is present in most user databases and provides a method of generating a serial field.

Table 2-19 tbl_unid

Column_name	Type	Length	Nulls	Description
unid_type	char	32	1	The unique ID type. You can maintain many different types of unid values with each generating a serial field by means of a suitable function.
unid	number	38	1	The current unid value for the unid type.
pad#	char	255	1	Where # is a number between 1 - 4. Forces each record to reside on a separate database page (allowing improved database concurrency).

Table indexes:

```
unid_type
```

tbl_unload_param

This table is used by the utilities. It provides information about the parameters inserted or deleted by the stored procedures.

Table 2-20 tbl_unload_param

Column_name	Type	Length	Nulls	Description
seq_no	Number	38	0	Sequence Number of the table
col_no	Number	38	0	Column Number of the parameter in the table
para_name	Varchar2	80	0	Name of the Parameter
default_flag	Number	38	0	Default flag
sp_type	char	3	0	Type of the stored procedure
rows_int	Number	38	0	Number of Rows

Table indexes:

seq_no, para_name, sp_type

This table will be available in the Control and SARM databases. It contains data relevant to the respective database it is in.

tbl_unload_sp

This table provides information about the stored procedures used to insert and delete data from ASAP tables. This table is used by utilities which modify these ASAP tables.

Table 2-21 tbl_unload_sp

Column_name	Type	Length	Nulls	Description
seq_no	Number	38	0	Sequence Number of the table
Tbl_name	Varchar2	40	0	Name of the ASAP table
New_sp	Varchar2	40	0	Stored Procedure used to insert the data into the table
Del_sp	Varchar2	40	0	Stored Procedure used to delete data from the table
List_sp	Varchar2	40	0	Stored Procedure used to list the data in the table

Table indexes:

seq_no

This table will be available in the Control and SARM databases. It contains data relevant to the respective database it is in.

SARM database

This section details the SARM database tables.

Work order audit information

This section describes the various methods of extracting work order audit information from the following tables:

- tbl_wrk_ord – contains the current status of work orders.
- tbl_info_parm – contains information parameters that are returned to the SRP from the NEP State Tables.
- tbl_srq_log – contains information pertaining to the system's interaction with NEs and resulting changes to work order status and switch history information of the NE responses.
- tbl_wo_audit – tracks the work order's status as it is processed. The level of information captured in tbl_wo_audit is governed by the ASAP configuration variable WO_AUDIT_LEVEL.

The level of logging for tbl_srq_log and tbl_wo_audit is governed by the ASAP.cfg configuration parameter WO_AUDIT_LEVEL. The following list explains the available audit levels:

- 0 – no auditing occurs. No information is placed in the tbl_wo_audit table.
- 1 – there is one audit entry per work order in the tbl_wo_audit table as it is tracked through the system.
- 2 – provides all functions of level 1 plus the audit level entries for all error states in the tbl_wo_audit table.
- 3 – provides all functions of level 2 plus it tracks the provisioning of a work order through the entire provisioning process in the tbl_wo_audit table. For example, when the ASDL was started, when it was placed in pending queue, where in the pending queue it is, when it was sent to the NEP etc.
- 4 – all events are inserted into the tbl_wo_audit table. This level is intended to debug the work order auditing process.

If the WO_AUDIT_LEVEL configuration parameter is set to 2 or greater, SRQ_ERROR_EVENTS are written to tbl_srq_log. Whenever it is set less than 2, WO_AUDIT_LEVEL events are not written into this table.

Events of other types, such as NE_RESP_EVENTs, NE_CMD_EVENTs and SRQ_INFO_EVENTs, are written into tbl_srq_log even when WO_AUDIT_LEVEL=0. These events track the information from the NEP's perspective, including:

- response logging (commands sent to and responses received from the NE)
- asdl_exit types and associated error messages
- information such as when a command was sent to the NE.

Viewing work order audit information

The OCA client provides a work order audit function that displays events related to the processing of work orders contained in tbl_wo_audit. These events include, but are not limited to, the event text and the data and time of these events.

The asap_utils utility provides additional functions that are particularly related to the processing of work orders.

- **1. SARM - Service Requests in DB** – lists the SRQs currently resident in the ASAP database. It details the order ID, status, priority, due date, parent order, batch group
- **2. SARM - In Proc Requests Summary** – lists the number of requests currently in progress in the ASAP database (this is determined to be orders in a Loading or In Progress state).
- **3. SARM - In Proc Requests Details** – lists details of any work orders currently in progress within the SARM.
- **4. SARM - Work Order Queue Summary** – lists the number of orders in each of the SARM order queues. The queues include:
  - Ready Queue – orders currently in progress.
  - Rollback Queue – orders currently being rolled back.
  - Auto Held Queue – orders that are being held by the SARM and not released for some reason
- **5. SARM - Work Order Queue Details** – provides the order details of each order in the SARM work order queues. Such queues are global to the SARM.

- **6. SARM - Work Order Lock States** – lists the orders in progress and their respective lock states. Generally, an in progress order will have a local lock. Only in the high availability configuration will orders be remotely locked.
- **7. SARM - ASDL/NE Queue Summary** – provides summary details about each NE in the system including:
  - the NE, technology and software load
  - the NEP managing the NE
  - the current state of the NE
  - Down, Connecting, Available, Maintenance, Disabled
  - the time estimate (sec.) for ASDL processing to that NE
  - the number of ASDLs pending to that NE (in a prioritized queue), the number currently in progress, the number of connections open to that NE, and the number of ASDLs waiting to be retried to that NE.
- **8. SARM - ASDL/NE Queue Details** – provides details about each ASDL in the Pending, In Progress and Retry ASDL queues to each NE in the system.

A complete description of these and other asap_utils functions is located in the *ASAP System Server Configuration Guide*.

The level of logging is governed by the ASAP.cfg configuration parameter WO_AUDIT_LEVEL. The following list explains the available audit levels:

- 0 – no auditing occurs. No information is placed in the tbl_wo_audit table.
- 1 – there is one audit entry per work order in the tbl_wo_audit table as it is traced through the system.
- 2 – provides all functions of level 1 plus the audit level entries for all error states in the tbl_wo_audit table.
- 3 – provides all functions of level 2 plus it tracks the provisioning of a work order through the entire provisioning process in the tbl_wo_audit table. For example, when the ASDL was started, when it was placed in pending queue, where in the pending queue it is, when it was sent to the NEP etc.
- 4 – all events are inserted into the tbl_wo_audit table. This level is intended to debug the work order auditing process.

If the WO_AUDIT_LEVEL is set to 2 or greater, SRQ_ERROR_EVENTS are written to tbl_srq_log. Whenever it is set less than 2, WO_AUDIT_LEVEL events are not written into this table.

Events of other types, such as NE_RESP_EVENTs, NE_CMD_EVENTs and SRQ_INFO_EVENTs, are written into tbl_srq_log even when WO_AUDIT_LEVEL=0. These events track the information from the NEP's perspective, including:

- response logging (commands sent to and responses received from the NE)
- asdl_exit types and associated error messages
- information such as when a command was sent to the NE.

SARM database tables

This section details the SARM database tables.

tbl_asap_srp

This static ASAP database table defines all Service Request Processors (SRPs) currently configured in the ASAP system. Any ASAP application process that communicates with the SARM as an SRP using the SRP API must be defined in this table.

Upon start up, the SARM opens one or more network connections to each SRP defined in this table. For each SRP, you can configure the SARM to issue an ASAP system event with each work order notification transmitted back to the SRP. The following are possible system events:

- NO EVENT – No event is sent to the SRP.
- NULL – The event is sent to the SRP or no system event is generated.
- Others – The event is sent to the SRP or the system event is generated of the type specified. This allows the SARM to be configured by the SRP and not globally as it would if this was performed by means of a configuration variable.

For more information on event types, see "tbl_event_type."

The Java SRP must be configured to be a socket type connection.

You are responsible for populating and maintaining this table.

You can maintain this table in the following ways:

- Using the Service Activation Configuration Tool (SACT). Refer to the *ASAP Server Configuration Guide* for instructions.
- Using stored procedures, including:
  - **SSP_new_srp** adds new SRP definitions.
  - **SSP_del_srp** deletes SRP definitions from this table.
  - **SSP_list_srp** lists the contents of this table.

Table 2-22 tbl_asap_srp Columns

Column_name	Type	Length	Nulls	Description
srp_id	varchar2	8	0	The logical SRP server name.
srp_desc	varchar2	255	1	A description of the SRP.
srp_conn_type	char	1	0	Connection protocol from the SARM to the SRP. Valid values are: • O – for Open Client. • S – for sockets.
srp_host_name	varchar2	80	1	Name of the machine that the SRP resides upon.
srp_host_port	varchar2	6	1	The port number that the SRP is waiting on for socket connections.

Table 2-22 (Cont.) tbl_asap_srp Columns

Column_name	Type	Length	Nulls	Description
wo_estimate_evt	varchar2	8	1	Work order estimate event. If you configure the SARM using the configuration variable WO_TIME_ESTIMATE_ON to perform this work order estimation calculation, the SARM determines the approximate number and processing time for ASDLs on the work order. The SARM then calculates an approximate time estimate for this work order.
wo_failure_evt	varchar2	8	1	Work order failure event.
wo_complete_evt	varchar2	8	1	Work order complete event.
wo_start_evt	varchar2	8	1	Work order start event.
wo_soft_err_evt	varchar2	8	1	This notification is generated whenever an ASDL response on the work order is 'Fail But Continue'. If the work order fails or completes, then this notification is followed by a Failure or Completion notification. If this event is not required by the SRP, you must specify NO_WO_EVENT_NOTIFICATION (NO EVENT).
wo_blocked_evt	varchar2	8	1	Not supported.
wo_rollback_evt	varchar2	8	1	Generated when a work order fails and rollback is configured on one or more CSDLs on the order. It is also generated when a completed work order is cancelled and the SARM rolls back any completed ASDLs on the work order. If this event is not required by the SRP, specify NO_WO_EVENT_NOTIFICATION (NO EVENT).
wo_timeout_evt	varchar2	8	1	Work order timeout event.
ne_unknown_evt	varchar2	8	1	If defined, the system event issued by the SARM when an NE unknown notification is being transmitted back to the SRP from the SARM. This notification is transmitted to the SRP upon a Remote to Host NE, or DN to Host NE routing error.
ne_avail_evt	varchar2	8	1	No longer used.
ne_unavail_evt	varchar2	8	1	No longer used.
wo_accept_evt	varchar2	8	1	If defined, the system event issued by the SARM when a work order acceptance notification is being transmitted back to the SRP from the SARM.

Table 2-22 (Cont.) tbl_asap_srp Columns

Column_name	Type	Length	Nulls	Description
aux_srp_id	varchar2	8	1	Specifies the auxiliary SRP to where the SARM opens network connections in the event that the primary SRP is unavailable. Set this field to NULL.
aux_srp_conn_type	char	1	1	Connection protocol for SARM to auxiliary SRP communications. Valid values are: • O – for Open Client. • S – for sockets.
aux_srp_host_name	varchar2	80	1	Name of the machine that the auxiliary SRP resides upon.
aux_srp_host_port	varchar2	6	1	The number of the port that the auxiliary SRP is waiting on for socket connections.

Table indexes:

`srp_id`

tbl_asap_stats

tbl_asap_stats provides a way of gathering and analyzing statistics related to the ASAP provisioning process.

If you enable the ASAP statistics gathering by using the NEP configuration variable ASAP_STATS_ON, the dynamic table will contain a log of all the ASAP statistical entries. Statistical entries are made by the Interpreter State Table action, LOG_STAT, and are passed to relevant data fields.

Table 2-23 tbl_asap_stats Columns

Column_name	Type	Length	Nulls	Description
stats_unid	number	38	0	The unique ID of the statistical entry which is generated at insert time.
stats_dts	date	-	0	The date and time that the statistical entry was logged.
wo_id	varchar2	80	0	The work order to which this statistical entry is related. This is determined from the work order table at insert time.

Table 2-23 (Cont.) tbl_asap_stats Columns

Column_name	Type	Length	Nulls	Description
wo_stat	number	1	0	The status of the associated work order. This is determined from the work order table at insert time. An update trigger updates the work order status to match that of the work order. This means that when a work order is completed, all statistical entries associated with that work order is updated to 'Completed'. The possible values for this field are the same as those in wo_stat in the SARM database table tbl_wrk_ord. See "tbl_wrk_ord (SARM)."
user_id	varchar2	64	0	The user ID associated with this log entry. The user ID is determined from the orig_login field in the SARM database table tbl_wrk_ord at insert time. See "tbl_wrk_ord (SARM)."
srq_id	number	38	0	The service request related to this statistical entry.
asdl_cmd	varchar2	80	0	The ASDL being provisioned when the statistics log entry was created.
mcli	varchar2	80	1	If set, the Remote NE.
hcli	varchar2	80	0	The Host NE to which the ASDL currently being processed is routed.
dn	varchar2	24	1	If set, the directory number currently being provisioned.
len	varchar2	16	1	If set, the line equipment number currently being provisioned.
ne_cmd	varchar2	255	1	If set, the NE command string sent to the NE.
cmd_type	varchar2	25	1	If set, the type of NE command string. The possible values which form the basis for simple statistical gathering include: • ADD – service addition. • REMOVE – service removal. • CHANGE – service change. • QUERY – query only, no provisioning performed. You can customize this field and also specify other command types to better suit your statistical requirements.

Table 2-23 (Cont.) tbl_asap_stats Columns

Column_name	Type	Length	Nulls	Description
cmd_stat	varchar2	25	1	If set, identifies the status of the NE command after transmission to the NE. The possible values which form the basis for statistical gathering include: • SUCCEED – Command completed successfully. • FAIL – Command failed. • RETRY – Command failed but retried. • MAINTENANCE – Command failed because the NE is currently unavailable to receive provisioning requests. • SOFT_FAIL – Command failed but processing continues on other ASDLs. • DELAYED_FAIL – An ASDL had failed during provisioning. The SARM skips any subsequent ASDL in the CSDL, continues provisioning at the next CSDL, and then fails the order. This field is customizable. You can specify other command states to better suit your statistical requirements. Refer to the *ASAP Cartridge Development Guide* for more detailed descriptions of these base_types.
cmd_reason	varchar2	255	1	If set, identifies the reason that the NE command failed. This is the error string returned from the NE. You can generate reports to determine the principal error conditions in the provisioning process.
parm1	varchar2	25	1	Multi-purpose text field in which customer specific information is logged. This field logs additional information in this table. To allow this in a flexible manner, these miscellaneous parameter fields have been provided.
parm2	varchar2	25	1	Same as parm1.
parm3	varchar2	25	1	Same as parm1.
parm4	varchar2	25	1	Same as parm1.
parm5	varchar2	25	1	Same as parm1.
parm6	varchar2	25	1	Same as parm1.

Table indexes:

```
Indexes

stats_unid
wo_id
```

```
wo_stat, stats_dts
cmd_stat, stats_dts
```

tbl_asdl_config

This static table defines the ASDL configuration information required to handle routing and rollback at the ASDL level. It is used by the SARM to determine which NEP to route the command to, whether rollback is required for this ASDL, and if so, the rollback ASDL to use. You are responsible for populating and maintaining this table.

- **SSP_new_asdl_defn** adds new ASDL configuration information to this table.
- **SSP_del_asdl_defn** deletes ASDL configuration information from this table.
- **SSP_list_asdl_defn** lists the contents of this table.

Table 2-24 tbl_asdl_config Columns

Column_name	Type	Length	Nulls	Description
asdl_cmd	varchar2	80	0	The ASDL command.
reverse_asdl	varchar2	80	0	Only required if rollback is configured on the ASDL. If the ASDL requires rollback, the reverse ASDL command must be invoked.
ignore_rollback	char	1	0	Ignore rollback flag. Possible values are: • Y – Rollback is ignored for this ASDL. • N – Rollback is required for this ASDL. In this case, rollback is only performed if there is a valid rollback ASDL command defined. Rollback can be initiated for this ASDL in two cases. • The work order fails and rollback is configured on one or more CSDLs on the work order. In this case, if configured, this ASDL is rolled back. • When a cancellation is received on the work order. If this ASDL has been completed and rollback is configured, the SARM initiates rollback for this ASDL.
route_flag	char	1	0	Routing of the ASDL. Possible value is: • (N) ROUTE_TO_NEP – The ASDL is routed to the NEP only. This value is defined in the header file sarm_defs.h. Deprecated. Column maintained for backward compatibility only.
description	varchar2	255	1	A description of the ASDL.
asdl_timeout	number	8	0	The maximum number of seconds for an ASDL timeout. If there is no response in the given timeout value, the ASDL will timeout. For more information, see the *ASAP System Administrator's Guide*.

Table 2-24 (Cont.) tbl_asdl_config Columns

Column_name	Type	Length	Nulls	Description
asdl_retry_number	number	8	0	The maximum number of retries if an ASDL requests timeout. If the number of retries exceeds asdl_retry_number, the order is failed and rolled back. For more information, see the *ASAP System Administrator's Guide*.
asdl_retry_interval	number	8	0	The time period in seconds between ASDL retries. For more information, see the *ASAP System Administrator's Guide*.

Table indexes:

`asdl_cmd`

tbl_asdl_log

This dynamic table is used by the SARM to log the ASDLs as they are sent to the NEP to facilitate rollback.

Table 2-25 tbl_asdl_log Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	The service request ID uniquely identifying where the ASDL belongs.
asdl_stat	number	1	0	Status of the ASDL. If defined, this field is updated while processing the ASDL and its rollback ASDL. Possible values include: • ASDL_NEP_COMPLETE 10 • ASDL_NEP_FAIL 11 • ASDL_NEP_RBACK_COMP 12 • ASDL_NEP_RBACK_FAIL 13 • ASDL_INITIAL 14 • ASDL_NEP_FAIL_CONTINUE 17 • ASDL_NEP_FAIL_DELAYED 20 These values are defined in sarm_defs.h.
asdl_unid	number	38	0	A unique ID identifying the ASDL log entry. This is a unique identifier and is assigned sequentially starting at the value1.
csdl_seq_no	number	38	0	The sequence number uniquely identifying the CSDL on the SRQ that generated this ASDL.
asdl_option	varchar2	255	1	Reserved.
asdl_cmd	varchar2	80	0	The ASDL command.

Table 2-25 (Cont.) tbl_asdl_log Columns

Column_name	Type	Length	Nulls	Description
rollback_asdl	varchar2	80	1	If rollback is required on this ASDL, use this command to roll back the original ASDL. It is set by the SARM once the rollback is completed as a history record. This field is NULL unless rollback of this ASDL is performed.
comp_dts	date	-	1	The completion date and time of the ASDL processing.
rollback_dts	date	-	1	The date and time of the ASDL rollback.
host_clli	varchar2	80	0	The Host NE to which the ASDL is routed by the SARM.
queue_dts	date	-	1	The date and time when the ASDL was placed in the SARM provisioning queue.
start_dts	date	-	1	The date and time when provisioning starts for this ASDL. The difference between this value and the queue_dts represents the time the ASDL spent in the queue of pending ASDLs before being transmitted to the NEP for provisioning.
retry_count	number	38	1	A count of the number of times the ASDL was retried at the NE.

Table indexes:

```
srq_id, asdl_stat, asdl_unid
```

tbl_asdl_parm

A static table used by the SARM to define the parameter labels and values associated with a given ASDL. It also provides the mapping between the CSDL parameter labels received from the SRP (csdl_lbl) and the ASDL parameter labels (asdl_lbl) transmitted to the NEP for interpretation by the Interpreter State Tables.

For each CSDL label (csdl_lbl), the SARM checks the current CSDL parameter name value pairs for a matching label. If no matching label is found, it checks for a label in the work order global parameter name value pairs. If no matching label is found in either of these parameter name value pairs and the parameter type (param_typ) which is mandatory, the default value (default_vlu), is used.

If no default value is set, the SARM registers an ASDL parameter mapping failure. If the parameter is Indexed, the csdl_lbl must contain a “++" or the SARM will not start. There is considerable translation mapping logic related to both the tbl_csdl_asdl and this table.

> **Note:**
>
> By convention, the ++ notation appears at the end of the label within square brackets. This convention makes it easy to identify the index.

An example of this translation is contained in the *ASAP Cartridge Development Guide*. Refer to the section on dynamic routing scenarios.

You are responsible for populating and maintaining this table.

- **SSP_new_asdl_parm** adds ASDL parameters to tbl_asdl_parm.
- **SSP_del_asdl_parm** adds ASDL parameters to tbl_asdl_parm.
- **SSP_list_asdl_parm** lists the contents of this table.

Table 2-26 tbl_asdl_parm Columns

Column_name	Type	Length	Nulls	Description
asdl_cmd	varchar2	80	0	The ASDL command.
parm_seq_no	number	38	0	The sequence number of the ASDL parameter within the parameter list.
asdl_lbl	varchar2	80	0	The ASDL label used in transmitting the ASDL parameter name value pairs to the NEP for interpretation by the NE State Tables or Java methods.
csdl_lbl	varchar2	80	0	The CSDL or global work order parameter label, which is transmitted to the SARM by the SRP or returned by the NEP State Tables or Java methods as a return CSDL global parameter. For an indexed or compound parameter, the CSDL command defined in this table is not exactly the same as the label transmitted by the SRP. This table stores the CSDL's base name.
default_vlu	varchar2	255	1	In the case when the CSDL parameter is not passed to the SARM, the SARM substitutes the default value for the csdl_lbl. This default parameter value is referenced only for Scalar parameters.

Table 2-26 (Cont.) tbl_asdl_parm Columns

Column_name	Type	Length	Nulls	Description
param_typ	char	1	0	There are three ASDL parameter formats: • SCALAR – Specifies the parameter label transmitted on the ASDL command. • COMPOUND – Specifies the base name of the Compound parameter transmitted on the ASDL command. • INDEXED – Specifies the base name of the ASDL command transmitted on the ASDL command. ASDL parameters can be either required or optional. Consequently, the possible parameter type values include: • R – required scalar parameter. • O – optional scalar parameter. • C – required compound parameter. This parameter type is also used for compound indexed parameters. For more information, see the *ASAP Cartridge Development Guide*. • N – optional compound parameter. • M – mandatory indexed parameter. • I – optional indexed parameter. • X – required XML parameter. • Y – optional XML parameter. • P – required XPATH parameter. • Q – optional XPATH parameter. • S – Parameter count. This value gives the State Table or Java method the total number of parameters associated with this ASDL command. • + – Current index value for this ASDL. Only applicable to indexed ASDLs. For more information about XML and XPATH parameters, see *ASAP Cartridge Development Guide*.
dep_asdl_lbl	varchar2	80	0	An ASDL parameter that identifies the XML document that the XPATH is evaluated against. Applies only to parameter types P and Q.

Table indexes:

```
asdl_cmd, parm_seq_no
```

For details and examples of various parameter types, refer to the *ASAP Cartridge Development Guide*.

tbl_asdl_response

In this ASDL Loopback testing table, the Interpreter must be placed in loopback mode by means of the Interpreter configuration parameter LOOPBACK_ON.

In loopback mode, when an ASDL completes successfully from the State Table, the Interpreter refers to this table to determine the action to take before completing the Interpreter ASDL invocation.

This table specifies a time interval to wait, (useful to test time-out conditions), for various State Table exit states and ASDL states returned by the Interpreter processing. You only need to perform the test to populate this table when loopback testing is being performed.

Table 2-27 tbl_asdl_response Columns

Column_name	Type	Length	Nulls	Description
asdl	varchar2	80	0	The ASDL for which loopback processing is provided.
parm_option	varchar2	80	0	Reserved for future use.
exit_status	varchar2	20	0	The exit status returned to the Interpreter from the simulation. The possible values include: • MAINT – Simulates the NE having gone into Maintenance mode and allows testing of the conditions. • FAIL – Simulates a hard error from the State Table. • SUCCEED – Default successful State Table execution. • STOP – Stops a work order after the execution of a state table or script is complete.

Table 2-27 (Cont.) tbl_asdl_response Columns

Column_name	Type	Length	Nulls	Description
asdl_status	varchar2	20	1	The ASDL status returned to the SARM. This column is only used when exit_status is FAIL. The possible values include: • RETRY – The ASDL is retried by the SARM. You can configure both the retry period and number of retries within the SARM using the configuration variables (NUM_TIMES_RETRY and RETRY_TIME_INTERVAL). • CONT or SOFT_FAIL – The ASDL fails but provisioning continues, for example, attempting to remove a service that is not present. • DELAYED_FAIL – The ASDL fails but provisioning on the rest of the order continues, at the end of which the order will fail. • FAIL – The ASDL fails and all provisioning on this work order ceases, for example, an unrecoverable error occurred. • STOP – Stops a work order after the execution of a state table or script is complete. Refer to the *ASAP Cartridge Development Guide* for more detailed descriptions of these base_types.
asdl_time	number	38	0	The time interval, in seconds, that the Interpreter waits before returning a response to the invoking server. This interval is used to simulate a delay in the processing of the NE responses to mimic usual run-time conditions.
error_text	varchar2	255	1	An error text description sent back to the SARM detailing the cause of the error. Error text can be viewed by the front-end user interface. If it is not explicitly specified, generic descriptive text is supplied.

Table indexes:

`asdl, asdl_option`

tbl_aux_wo_prop

tbl_aux_wo_prop is a class A dynamic table that serves as an extension to tbl_wrk_ord. tbl_aux_wo_prop was designed to accommodate additional pre-defined work order properties to supplement the ones contained in tbl_wrk_ord. Currently, the only extended property supported in tbl_aux_wo_prop is WO_SECURITY_PROP. This property is maintained for each work order. If WO_SECURITY_PROP = 0, then work order information is eligible to be

output to diagnostic files. If WO_SECURITY_PROP = 1, then no work order information is written to diagnostic files.

Refer to the *ASAP System Administrator's Guide* for more information on secure work order information.

When the work orders in the tbl_wrk_ord are deleted, the corresponding records in tbl_aux_wo_prop are deleted. This table can be purged using function **SSP_db_admin**. For guidelines and instructions on database purging, refer to the *ASAP System Administrator's Guide.*

Table 2-28 tbl_aux_wo_prop Columns

Column Name	Type	Length	Nulls	Nulls
wo_id	varchar2	80	0	The ID of the work order property.
name	varchar2	80	0	The name of the work order property.
value	varchar2	255	1	The value given to the work order property.

Table indexes:

```
Non-unique

wo_id
```

tbl_blackout

This static table contains date and time periods for which a particular Host NE is deemed unavailable by the user for ASAP updates.

The NEP State Table reads this table using the current time. If the current time is within a blackout period, the State Table returns a maintenance mode condition, therefore, disconnects from the Host NE. This process is continually retried until the Host NE is no longer blacked out. You can use this table when there are systems competing for limited ports on the Host NE. You are responsible for populating and maintaining this table.

Note that if the dayname entry is present, then the configuration is considered to be Static. The implementation will check for the start time and end time alone, and assume that both fall on the same day. When configuring a blackout period that spans from one day to the next (e.g. from 22:00 until 01:00 the next day) you must configure two separate lines in tbl_blackout: one for 22:00:00 until 00:00:00 and one from 00:00:00 until 01:00:00.

Table 2-29 tbl_blackout Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE that is blacked out.

Table 2-29 (Cont.) tbl_blackout Columns

Column_name	Type	Length	Nulls	Description
dayname	varchar2	10	1	If this field contains the valid name of a day of the week (for example, Sunday or Monday) then the date portions of start_tm and end_tm are ignored. If this field is empty than both the date and time portions from start_tm and end_tm will be used. If this field contains an invalid value, the blackout will not occur.
start_tm	date	-	0	The start date and time for the blackout interval.
end_tm	date	-	0	The end date and time for the blackout interval.
description	varchar2	255	1	A description of the NE blackout reason.

Table indexes:

```
host_clli, start_tm, dayname
```

tbl_clli_route

This static table contains the mapping between a Remote NE and its Host NE. It is used to determine the Host NE when a Remote NE is provided on the work order. Given the Remote NE and ASDL, this table provides the Host NE to which this ASDL is routed.

If no Remote NE is present on the work order, the table tbl_nep_rte_asdl_nxx is used to route the ASDL by DN. This is only performed if the parameter MCLI is present on the ASDL and is set to the special literal SKIPCLLI. You are responsible for populating and maintaining this table.

- **SSP_new_clli_map** adds new CLLI-to-Host CLLI mapping definitions to tbl_clli_route.
- **SSP_del_clli_map** deletes CLLI-to-Host CLLI mapping definitions in tbl_clli_route.
- **SSP_list_clli_map** lists the contents of this table.

Table 2-30 tbl_clli_route Columns

Column_name	Type	Length	Nulls	Description
mach_clli	varchar2	80	0	Remote NE.
host_clli	varchar2	80	0	Host NE to which the Remote NE is connected.
asdl_cmd	varchar2	80	1	The ASDL command for which this routing definition is valid. This is useful for services that require ASDL-dependent routing.

Table indexes:

```
mach_clli, asdl_cmd
```

tbl_comm_param

This static table contains communication parameters required to communicate with various external systems. It is possible to specify such communication parameters by device type, for example, serial, TCP/IP Telnet, Generic, etc.

You can also specify communication parameters for:

- Host NEs and all devices.
- Host NEs and a particular device.
- Specific Host NE and all devices.
- Specific Host NE and a specific device.

If an entry exists for the Host NE and device of a particular Command processor in the NEP, it overrides any of the previous entries. You can also specify parameters in this table for particular Host NEs and devices or combinations.

Each parameter is made available to the State Table or Java method. tbl_comm_param specifies various parameters that are specific to the Host NE or device. It allows State Tables or Java methods to employ processing that is specific to the Host NE or device, where required. You are responsible for populating and maintaining this table.

- **SSP_new_comm_param** adds new communication parameters for a specified device type, host, and device into tbl_comm_param.
- **SSP_del_comm_param** deletes parameter information from tbl_comm_param.
- **SSP_list_comm_param** lists the contents of this table.
- **SSP_get_async_ne** lists NEs which have an ASYNC_CONN communication parameter defined with a value of TRUE or FALSE.

Table 2-31 tbl_comm_param Columns

Column_name	Type	Length	Nulls	Description
dev_type	char	1	0	The device line type. The possible values include: • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection
host	varchar2	80	0	The Host NE for which this parameter value applies. If specifying communication parameters for all Host NEs, set this parameter to COMMON_HOST (COMMON_HOST_CFG).

Table 2-31 (Cont.) tbl_comm_param Columns

Column_name	Type	Length	Nulls	Description
device	varchar2	40	0	The device for which this parameter value applies. If specifying communication parameters for all devices, set this parameter to COMMON_DEVICE (COMMON_DEVICE_CFG).
param_label	varchar2	80	0	The parameter label. There are many communication parameters that you must specify in this table in order for the ASAP communication library to function correctly. These communication parameters and their usage are detailed in the *ASAP System Administrator's Guide*.
param_value	varchar2	255	0	The parameter value.
param_desc	varchar2	255	1	Description of the communication parameter.

Table indexes:

```
dev_type, host, device, param_label
```

tbl_cp_mux

Specifies the mapping between command processor logical devices and NEP multiplexing devices.

Table 2-32 tbl_cp_mux Columns

Column_name	Type	Length	Nulls	Description
cp_device	varchar2	40	0	A logical device used in the communications between a command processor thread and a multiplexing device. This device must be first configured in the table tbl_resource_pool.
mux	varchar2	40	0	The name of the multiplexing device that the specified command processor logical device maps to. The multiplexing device must be first configured in the table tbl_nep_mux.

Table indexes:

```
cp_device
```

tbl_csdl_asdl

This static table used by the SARM to map and sequence CSDL commands to ASDL commands. For each ASDL associated with a CSDL, SARM checks if the ASDL is valid for the CSDL. The final determination of whether the ASDL is valid depends on the ASDL parameter translation process specified by tbl_asdl_parm. To perform this translation, certain conditions must apply. These conditions are identified in columns cond_flag and eval_exp. An ASDL is valid only if both conditions are satisfied.

Examples of various configurations and instructions on configuring CSDL-to-ASDL translation are described in the *ASAP System Administrator's Guide*.

You are responsible for populating and maintaining this table using one of the following mechanisms:

- using XML schemas and deploying these services using the Service Activation Deployment Tool (SADT). Refer to the *ASAP Cartridge Development Guide* for more information.
- using stored procedures
  - **SSP_new_CSDL_asdl** adds a new CSDL-to-ASDL mapping definitions to tbl_csdl_asdl
  - **SSP_del_CSDL_asdl** deletes CSDL-to-ASDL mapping definitions from tbl_csdl_asdl
  - **SSP_list_csdl_asdl** lists the contents of this table.

Table 2-33 tbl_csdl_asdl Columns

Column_name	Type	Length	Nulls	Description
csdl_cmd	varchar2	80	0	CSDL command that translates into one or more ASDL commands.
asdl_seq_no	number	38	0	The sequence number of the ASDL command associated with the CSDL. This enables one CSDL to generate more than one ASDL command.
asdl_cmd	varchar2	80	0	ASDL command to which the CSDL translates.
pnr	number	38	1	Values are: • 0 (default) – This ASDL is not the 'point of no return' for rollback purposes • 1 – This ASDL is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this ASDL but no further. • 2 – 'point of no return' for no rollback. Once past this ASDL, no rollback can occur.

Table indexes:

```
csdl_cmd, asdl_seq_no
```

tbl_csdl_asdl_eval

This table contains CSDL to ASDL mappings and the multiple condition expressions defined for the mappings

Table 2-34 tbl_csdl_asdl_eval Columns

Column_name	Type	Length	Nulls	Description
csdl_cmd	varchar2	80	0	Foreign key to tbl_csdl_asdl.csdl_cmd.
asdl_seq_no	varchar2	80	0	The sequence number of the ASDL command associated with the CSDL. This enables one CSDL to generate more than one ASDL command.
cond_flag	char	1	0	The conditions to be met for the ASDL to be executed for the CSDL. The values for this field are as follows: • A – Always execute the ASDL for this CSDL when the expression is NULL (no expression), when the expression contains the string TRUE, or when the expression is evaluated to be true. • E – Check for the associated label to be present for the CSDL parameter label/value pairs and check that its value is equal to the label value. Similar to "=" or LIKE in the algebraic expression. • D – Check that the label is defined. Similar to ISDEF operations in the algebraic expression. • N – Check that the label is not defined. Checks that the label is similar to NOTDEF. The condition is checked for each ASDL associated with the CSDL. If the condition is satisfied, the ASDL is added to the list of ASDLs for the CSDL. If not, the ASDL is not executed. Validation is performed in the parameter validation stage in referencing the table tbl_asdl_parm.
label	varchar2	80	1	The parameter label to test in the condition flag.
value	varchar2	255	1	The parameter value associated with the label which is tested in the condition flag.
eval_exp	varchar2	255	1	Contains combination of parameter names, operators, and values to which the parameters are compared.
apply_from	number	38	1	The first indexed ASDL that this rule should apply to: Valid range is from 1 to 9999. Must be less than or equal to the value specified in column **apply_to.** If is not specified, then this rule will be applied to any indexed ASDL up to and including the one specified in column **apply_to**.

Table 2-34 (Cont.) tbl_csdl_asdl_eval Columns

Column_name	Type	Length	Nulls	Description
apply_to	number	38	1	The last indexed ASDL that this rule should apply to: Valid range is from 1 to 9999. Must be greater than or equal to the value specified in column **apply_from.** If is not specified, then this rule will be applied to any indexed ASDL starting from the one specified in column **apply_from**.

tbl_csdl_config

This static table contains the static CSDL configuration information that determines if the CSDL is configured for rollback, its provisioning sequence, and whether a failure of the CSDL generates a system event. You are responsible for populating and maintaining this table.

- **SSP_new_csdl_defn** adds new CSDL definitions to tbl_csdl_config.
- **SSP_del_csdl_defn** deletes CSDL definitions from tbl_csdl_config.
- **SSP_list_net_elem** lists the contents of this table.

Table 2-35 tbl_csdl_config Columns

Column_name	Type	Length	Nulls	Description
csdl_cmd	varchar2	80	0	The CSDL command.
rollback_req	char	1	0	A flag indicating if rollback is required for this CSDL. In certain circumstances, it may be preferable for no rollback to occur. If any of the CSDLs on a work order require rollback, the dynamic work order structure in SARM's memory is flagged and the entire work order is rolled back if any CSDL fails. Possible values are: • Y – Rollback required. • N – No rollback required. In the SRP database, **wo_rback** must be set to Y or D for the CSDL to rollback.

Table 2-35 (Cont.) tbl_csdl_config Columns

Column_name	Type	Length	Nulls	Description
csdl_level	smallint	1	0	The level of the CSDL in the SRQ. An integer between 0 and 255 that indicates the sequence level for the CSDL command within the work order. The SARM uses this integer to determine the order in which to provision CSDL commands from an SRP. The SARM then provisions CSDL commands that have lower level numbers first. Sequence levels are only relevant for inter-dependent CSDL commands As the SARM receives the CSDLs from the SRP, it reorders them in a sequence that corresponds to their respective CSDL levels to ensure the correct provisioning sequence. This reordered sequence may not be the same as the sequence in which the SRP transmitted the CSDLs to the SARM. The field csdl_id in the table tbl_srq_csdl contains the SRP CSDL sequence of the CSDLs transmitted to the SARM.
fail_event	varchar2	8	1	If set, the system event to be triggered upon CSDL failure. This event can log or print error messages, trigger system alarms, etc. Use this column to immediately notify personnel of CSDL failure.
complete_event	varchar2	8	1	If set, the system event triggered upon CSDL completion.
option_asdl	number	38	0	Reserved for future use.
description	varchar2	255	1	Description of the CSDL command. This description field is displayed in front-end user interfaces only.

Table indexes:

`csdl_cmd`

tbl_err_threshold

This static table specifies the threshold for the number of consecutive hard errors for a particular Host NE and ASDL. If the threshold is exceeded, the SARM requests the NEP to disable the specified Host NE. Such hard errors can either be generated directly by the State Tables or from user-defined exit types that map to hard errors. You are responsible for populating and maintaining this table.

- **SSP_new_err_threshold** adds a new threshold for a specific NE and ASDL command in tbl_err_threshold.
- **SSP_del_err_threshold** deletes a threshold from tbl_err_threshold.
- **SSP_list_err_threshold** lists the contents of this table.

Table 2-36 tbl_err_threshold Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE.
asdl_cmd	varchar2	80	0	The ASDL commands to be used to track consecutive hard errors.
threshold	number	38	0	The threshold number of consecutive hard errors that must occur for the SARM to disable the specified Host NE.

Table indexes:

```
host_clli, asdl_cmd
```

tbl_event_dataset

This table contains sets of parameters to be returned for events. Note that multiple trigger criteria (stored in "tbl_event_template") can map to a single set of return parameters. Also, the table can contain multiple rows for given **template_name**, with each row defining a particular parameter.

Table 2-37 tbl_event_dataset Columns

Column_name	Type	Length	Nulls	Description
template_name	varchar2	20	0	The name of the event template dataset. This is a unique user-defined identifier for a distinct return parameter dataset.
parameter_type	varchar2	20	0	Identifies the type of the parameter to be added to the event template dataset: • GLOBAL_PARAMETER • CSDL_PARAMETER • INFO_PARAM • EXTENDED_PROPERTY
csdl	varchar2	80	1	The CSDL to apply the event template dataset to. The csdl parameter is only required if the **parameter_type** is CSDL_PARAMETER or INFO_PARAM:
parameter_name	varchar2	20	0	Name of the parameter to be added to the event template dataset.

Table indexes:

```
template_name, parameter_type, cscl, parameter_name
```

tbl_event_template

This table maps event template names to event type / CSDL combinations. The **name** field maps to event template entries in tbl_event_dataset which detail extended event information to be returned in the work order header.

See "tbl_event_dataset" for more information.

Table 2-38 tbl_event_template Columns

Column_name	Type	Length	Nulls	Description
name	varchar2	20	0	The name of the event template. Points to a set of return parameters in tbl_event_dataset thus allowing for the case where multiple trigger criteria could map to a single set of return parameters.
event_type	varchar2	20	0	(Mandatory) The event type to apply the event template name to. Following are the event types: • Order Startup event: Returns order parameters, extended work order property. • Order Complete event: Returns order parameters, information parameters, extended work order parameters, and service action parameters. • Order Timeout event: Returns order parameters, information parameters, extended work order parameters, and service action parameters. • Order Fail event: Returns order parameters, information parameters, extended work order parameters, and service action parameters. **Note:** For each service action or just for the given service action, service action parameters are returned.
csdl	varchar2	80	1	The CSDL to apply the event template to. Optional. **Note**: This field is applicable only when the event type is set to **orderFailEvent**.

Table indexes:

```
event_type, csdl
```

tbl_ext_method_lib

This table provides the external method library details required by the SRT.

Table 2-39 tbl_ext_method_lib Columns

Column_name	Type	Length	Nulls	Description
name	varchar2	80	0	Name of the library
type	char	1	0	Type of the library • S - represents Script • Q - represents SQL • J - represents Jar
library	blob	-	1	The binary form of the library.

tbl_host_clli

This static table contains the Host NE, technology, and software load of each Host NE in the ASAP system. It also contains records for each Host NE to which the NEPs interface. You are responsible for populating and maintaining this table

- **SSP_new_net_host** adds new network element definitions to this table.
- **SSP_del_net_host** deletes network element definitions from this table.
- **SSP_list_net_host** lists the contents of this table.

Table 2-40 tbl_host_clli Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE identifier.
tech_type	varchar2	16	0	Technology of the Host NE or SRP.
sftwr_load	varchar2	16	0	Software version of the Host NE, for example, BCS33, PAC4, or the token software load of the SRP. The technology and software load must be consistent with the values given in the table tbl_nep_asdl_prog in order to ensure that the State Table Interpreter or JInterpreter are able to translate an ASDL.

Table indexes:

`host_clli`

tbl_id_routing

This static table is a routing database table that defines the mapping between an ASDL and the Host NE when using ID_ROUTING as an ASDL routing scheme.

- **SSP_new_id_routing** adds a host NE and the ID_ROUTING mapping record to tbl_id_routing.
- **SSP_del_id_routing** deletes a host NE and the ID_ROUTING mapping record from tbl_id_routing.
- **SSP_list_id_routing** lists the contents of this table.

Table 2-41 tbl_id_routing Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE which the ASDL command is routed to.
asdl_cmd	varchar2	80	1	The ASDL command that is being routed.
id_routing_from	varchar2	255	0	The starting point of a range of ID_ROUTING.
id_routing_to	varchar2	255	0	The end point of a range of ID_ROUTING.

If a CSDL maps to multiple ASDLs that are routed to different host NEs, you must ensure that tbl_ne_routing contains an entry for each ASDL. Each entry references a different host NE. For example:

Table 2-42 ID Routing example

fromRange	toRange	asdl_cmd	host_clli
100	200	A-ADD_POTS_LINE	HOUSTON
100	200	A-OPTION_ON	DALLAS

The following example shows how ID_ROUTING operates for IP addresses:

Table 2-43 ID Routing example

fromRange	toRange	asdl_cmd	host_clli
10.9.1.0	10.9.10.255	A-ADD_POTS_LINE	TORONTO
10.9.11.0	10.9.18.255	A-ADD_POTS_LINE	DALLAS

In this case, if the ASDL references an IP address of 10.9.3.25, the ASDL would be routed to the TORONTO host NE. The IP address 10.9.12.255 would be routed to DALLAS.

Refer to $ASAP_BASE\samples\ASDL_ROUTE for a sample ID routing based on IP addresses.

Table index:

```
host_clli
```

Once the SARM starts, tbl_id_routing is loaded into memory from the SARM database.

For exact matching, the value of id_routing_from, and id_routing_to must be configured identically.

tbl_info_parm

This dynamic table contains information parameters that are returned to the SRP from the NEP State Tables. These parameters contain information that is returned to the requesting external system as Compound parameters. The SRP uses a query API function to retrieve the Information parameters associated with a particular work order.

Table 2-44 tbl_info_parm Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The work order ID.
parm_lbl	varchar2	80	0	The parameter label.
parm_group	varchar2	80	1	NE parameter group information returned to the Host system. This parameter group is set in the State Tables before the Information parameters are returned to the SARM. The SRP API routines specify parameters belonging to particular parameter groups in the Information parameter retrieval.

Table 2-44 (Cont.) tbl_info_parm Columns

Column_name	Type	Length	Nulls	Description
parm_vlu	varchar2	255	1	The value associated with the parameter label. All control characters x0 to x1F, except x9, xA, and xD (tab, cr and nl), are stripped from the event text after it is retrieved from the database and before used in the XML document. This behavior is in compliance with XML 1.0 specification.
csdl_seq_no	number	38	0	The CSDL sequence number in the work order that is being processed when the Information parameter is generated.
csdl_cmd	varchar2	80	0	The CSDL command being processed when the Information parameter is generated from the Interpreter State Table.
csdl_id	number	38	0	The ID of the CSDL being processed when the Information parameter is generated. This CSDL ID is the same as in table tbl_srq_csdl and enables the SRP to track which CSDL the information parameter belongs to.
vlu_hint	char	1	0	Indicates whether the parameter value is a real value or an XML document or XPath expression. Can be one of the following: • X - indicates a reference to XML document • P - indicates a reference to XPath expression • T - indicates a reference to large text (reserved for future use)

Table indexes:

```
wo_id, parm_lbl, parm_group
```

tbl_label_value

This dynamic table stores event subscription information from the old CORBA SRP clients to handle event servers between CORBA SRP client and server. The new OCA SRP does not use this table.

Table 2-45 tbl_label_value Columns

Column_name	Type	Length	Nulls	Description
key_1	varchar2	16	0	Work Order ID.

Table 2-45 (Cont.) tbl_label_value Columns

Column_name	Type	Length	Nulls	Description
key_2	number	-	1	Event type. Possible values are defined oca.idls as follows: • (1) ASC_WO_COMPLETE_EVT • (2) ASC_WO_FAILURE_EVT • (3) ASC_WO_COMPLETE_FAILURE_EVT • (4) ASC_WO_OTHER_EVT • (5) ASC_WO_ALL_EVT • (6) ASC_WO_SOFT_ERR_EVT • (8) ASC_WO_TIMEOUT_EVT • (9) ASC_WO_ROLLBACK_EVT • (10) ASC_WO_ACCEPT_EVT • (11) ASC_WO_STARTUP_EVT • (12) ASC_WO_ESTIMATE_EVT • (13) ASC_NE_UNKNOWN_EVT
label	varchar2	600	0	An external system ID that is passed with the Work Order. It is not used by the SARM, but it is used by the SRP for proper routing to upstream systems.
value	varchar2	128	1	A sister external system ID that is passed with the Work Order. It is not used by the SARM, but it is used by the SRP for proper routing to upstream systems.

Table indexes:

```
key_1, key_2, label
```

tbl_large_data

This table stores XML data, XPath expressions or other large data for future use by the SRT.

Table 2-46 tbl_large_data Columns

Column_name	Type	Length	Nulls	Description
ref_id	number	20	0	Reference ID returned by the Database for the XML Order Data.
data	blob	-	1	Raw XML Data, XPath Expression or other large data.

tbl_msg_convert

This static table contains the language and format of events logged by the SARM as it processes requests. Log messages are sent back to the Host system, and therefore, it is important they are in the user's native language. This table provides you with a mechanism to insert your own event text in your own language by populating this table with suitable data.

The core system provides the base event text in the language USA English. If you want to use another language, you are responsible for populating this table with the events in that language.

- **SSP_new_intl_msg** adds a new international message record to tbl_msg_convert.
- **SSP_del_intl_msg** deletes an international message from tbl_msg_convert.
- **SSP_list_intl_msg** lists the contents of this table.

Table 2-47 tbl_msg_convert Columns

Column_name	Type	Length	Nulls	Description
lang_cd	varchar2	3	0	The language code. This code determines the text message and format the SARM utilizes. The language code that the SARM uses is set by the SARM configuration variable LANGUAGE_OF_MSG. The default is USA.
msg_id	number	38	0	Unique message identifier for messages referenced within the SARM.
msg_type	char	1	0	This field specifies the type of message formatting. Possible values are: • (D) Dynamic – The SARM performs parameter substitution into the formatted string in the message field which allows dynamic customizing of the message text. • (S) Static – The SARM does not perform any parameter substitution; instead it uses the raw message in the next message field.
message	char	255	0	The message text. If the type field is S, this field resembles a printf() format string. If the type field is D, no parameter substitution is performed.
var_description	char	40	1	Description of the substitutable fields within the message.
wo_audit	char	1	1	The destination for this log message. Possible values are: • ASAP_LOG_SRQ (srq_log) • ASAP_LOG_WOA (work order audit) • ASAP_SRQWOA (both) • ASAP_LOG_NO (none)

Table indexes:

```
lang_cd, msg_id
```

tbl_ne_config

This static table, which is used by the SARM and the NEP, contains configuration information for each Host NE within ASAP. It also identifies the NEP that manages each Host NE.

This table is read by the SARM to determine the NEs managed by each NEP. This allows the SARM to route the ASDLs to the appropriate NEP that is managing the Host NE. It defines attributes that are particular to each Host NE regardless of the number and nature of the connections to that NE.

This table is read by the NEP to determine the NEs to be managed by that NEP and the Session Managers to be spawned to control all interaction with the NEs. It also specifies the primary resource pools of devices used to connect to the NEs.You are responsible for populating and maintaining this table.

- **SSP_new_net_elem()** adds new network element definitions to this table.
- **SSP_del_net_elem()** deletes network element definitions from this table.
- **SSP_list_net_elem()** lists the contents of this table.
- **SSP_set_ne_loopback()** updates the table when the loopback state is set to ON, OFF, or GLOBAL through asap_utils.
- **SSP_new_net_element()** creates an extra throughput column with the default value of 0.
- **SSP_set_ne_throughput()** sets the minimum time for a transaction in milliseconds on the NE.

Table 2-48 tbl_ne_config Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE. This is the name of the SessionManager thread within the NEP which manages all interaction with the Host NE and all connections to it. For dynamic routing, this is the name of the template.
nep_svr_cd	varchar2	8	0	The logical name of the NEP server that connects to the Host NE.
primary_pool	varchar2	8	1	The primary resource pool used by the NEP managing the Host NE. It determines the primary devices used to connect to the Host NE. The primary pool is dedicated to a specific NE. The Host NE Session Manager within the NEP uses entries in this primary pool before attempting to use entries in the NEP's secondary resource pool to connect to this Host NE, as defined in tbl_nep.

Table 2-48 (Cont.) tbl_ne_config Columns

Column_name	Type	Length	Nulls	Description
max_connections	number	2	0	The maximum number of concurrent connections allowed to the Host NE. This includes connections from both primary and secondary pools. For multiple dedicated Command Processors to a Host NE, configure the spawn threshold to 1 and the kill threshold to 0. This ensures that multiple Command Processors are used. For dynamic routing, this information can be provided on each order.
drop_timeout	number	2	0	The maximum provisioning activity idle time, in minutes, that must elapse before the Session Manager managing the Host NE disconnects the primary connection to the NE. Until this threshold is exceeded, the Session Manager managing the NE maintains a dedicated Command Processor with a dedicated connection to the NE when there are no pending ASDL requests (maximum idle time). This time-out interval only applies to the primary connection to the NE, as all auxiliary connections have reached their kill_threshold by this point. For a busy Host NE, this time interval should be set high as there is a high probability of an incoming ASDL for the Host NE in that period. However, this value should be set low for a less active Host NE as the resource pool device in use by the Command Processor may be required by another Session Manager to communicate with another Host NE. This is most likely the case when the device is from the NEP's resource pool (the secondary pool which is also accessible to all Session Managers). For dynamic routing, drop timeout information can be provided on each order.
throughput	smallint	1	0	NE instance throughput control – the minimum number of milliseconds a transaction takes per NE instance. The value 0 disables NE instance thoughput control.

Table 2-48 (Cont.) tbl_ne_config Columns

Column_name	Type	Length	Nulls	Description
spawn_threshold	number	2	0	The number of ASDL requests in the SARM's “ASDL Ready Queue" destined to the NE. When this number is exceeded, the SARM requests the NEP Session Manager to open a new auxiliary connection to the NE. As the number of ASDLs in the SARM ASDL Ready Queue continues to exceed this spawn threshold, the SARM continues to request the NEP establishment of auxiliary connections to the NE. For example, the spawn threshold to a particular NE is 10. Once the ASDL Ready Queue size reaches 11, the SARM requests the NEP to establish an auxiliary connection to the NE. If the ASDL Ready Queue size reaches 12, the SARM requests another auxiliary connection be established by the NEP, and so on. This spawn threshold value should always be above the kill threshold if multiple connections are required to a particular NE. For dynamic routing, spawn threshold information can be provided on each order.

Table 2-48 (Cont.) tbl_ne_config Columns

Column_name	Type	Length	Nulls	Description
kill_threshold	number	2	0	Upon receiving an ASDL completion, the SARM determines whether the current ASDL Ready queue size is less than the kill threshold. If so, it checks whether the ASDL was completed by an auxiliary connection. If so, the SARM transmits a disconnect request to the NEP Session Manager which disconnects the auxiliary connection. If the ASDL completion was performed on the primary connection, the SARM will not issue a disconnect request. For example, if the spawn_threshold is 10 and the kill threshold is 5, when the SARM receives an ASDL completion when its ASDL Ready queue size is 4 or less, it checks whether the ASDL completion was performed by an auxiliary connection. If so, it issues a disconnect request to the NEP Session Manager managing this NE. If the ASDL completion was not performed by an auxiliary connection, as would be the case when the spawn threshold was never exceeded, the SARM will not issue a disconnect request. For multiple dedicated connections, set the relevant number of connections in max_connections, the spawn threshold to 1, the kill threshold to 0, and the appropriate number of devices in the primary and secondary resource pools. For dynamic routing, kill threshold information can be provided on each order.
template_flag	Char ('Y','N')	1	0	Flag to indicate if this network element entry identifies a static NE (N) or a dynamic network element template (Y). The default value is 'N'.
loopback_on	Char ('Y', 'N', 'G')	1	0	Y: The NE is in loopback state regardless of the LOOPBACK_ON ASAP configuration parameter. N: The NE is not in loop back state regardless of the LOOPBACK_ON ASAP configuration parameter. G: The NE loop back state is determined by the LOOPBACK_ON ASAP configuration parameter. The default value is 'G'.
request_timeout	number	8	0	The maximum number of seconds for an NE request timeout. If there is no response in the given timeout value, the associated ASDL will timeout. For more information, see the *ASAP System Administrator's Guide*.

Table 2-48 (Cont.) tbl_ne_config Columns

Column_name	Type	Length	Nulls	Description
request_retry_number	number	8	0	The maximum number of retries, if the NE requests timeout. If the number of retries exceeds request_retry_number, the order is failed and rolled back. For more information, see the *ASAP System Administrator's Guide*.
request_retry_interval	number	8	0	The time period in seconds between NE retries. For more information, see the *ASAP System Administrator's Guide*.

Table indexes:

```
nep_svr_cd, host_clli
```

tbl_ne_event

This table contains NE-related log messages.

Table 2-49 tbl_ne_event Columns

Column_name	Type	Length	Nulls	Description
event_identity	number	20	0	Unique ID of this log message in this table.
host_clli	varchar2	80	0	The Host NE identifier if an NE or the SRP name if an SRP.
event_dts	date	-	0	The time when the event was generated.
state	varchar2	80	0	The current state of the NE. Possible states are: • Available • Down • Connecting • Maintenance • Port Failure • Disabled • Unknown NE State
evt_text	varchar2	255	0	Description of the event.

Table indexes:

```
Indexes

evt_identity
host_clli, event_dts
event_dts
```

tbl_ne_monitor

Maintains the status of all NEs.

> **Note:**
>
> The tbl_ne_monitor table has been deprecated from ASAP 4.6.x onwards due to impact on the performance and so NE state information is no longer populated in it by default. However, this functionality can be turned on when you set the variable `WO_TIME_ESTIMATE_ON` to **1** in the **ASAP.cfg** file. On turning this configuration variable on the SARM database table tbl_ne_monitor gets populated with the NE state information and a new event WO_ESTIMATE_EVT is added to the SARM database table tbl_wo_event_queue. Oracle recommends not to use this feature in production.

Table 2-50 tbl_ne_monitor Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE where the ASDL is routed by the SARM. Even though one Host NE is specified on the CSDL, different ASDLs on that CSDL can be routed to different Host NEs.
nep_svr_cd	varchar2	8	0	Name of the NEP server.
state	varchar2	80	0	The current state of the NE. Possible states are: • Available • Down • Connecting • Maintenance • Port Failure • Disabled • Unknown NE State
asdl_time_est	number	38	0	Estimated time to process an ASDL.
pending_count	number	38	0	Number of ASDLs pending.
in_progress_count	number	38	0	Number of ASDLs in progress.
connect_count	number	38	0	Number of connections to the NE.
retry_count	number	38	0	The number of times the ASDL was retried at the NE.
held_count	number	38	0	Number of ASDLs held.
consec_fail_count	number	38	0	Number of consecutive failures.
connect_dts	date	-	0	Date/time stamp of the last connection.
disconnect_dts	date	-	0	Date/time stamp of the last disconnect.
avail_dts	date	-	0	Date/time stamp of the last time mode available.
maint_dts	date	-	0	Date/time stamp of the last time it was put into maintenance mode.

Table 2-50 (Cont.) tbl_ne_monitor Columns

Column_name	Type	Length	Nulls	Description
err_disable_dts	date	-	0	Date/time stamp of the last time an error disabled the NE.
err_enable_dts	date	-	0	Date/time stamp of the last time the NE was enabled after an error.
adm_disable_dts	date	-	0	The last time an administrator disabled the NE.
adm_enable_dts	date	-	0	The last time an administrator enabled the NE.
swd_sessions	number	38	1	Number of switch direct sessions.
swd_start_dts	date	-	1	The last time a session started.
swd_end_dts	date	-	1	The time when a switch direct session ended.
swd_user_id	varchar2	64	1	Switch direct user.
pad1	char	255	0	Padding to make a table row occupy a page. This reduces concurrence on the database data page by different database processes.
pad2	char	255	0	Same as pad1.
pad3	char	255	0	Same as pad1.
pad4	char	255	0	Same as pad1.

Table indexes:

`host_clli`

tbl_ne_strsub

Enables you to configure substitutions for unknown or unwanted control characters.

Table 2-51 tbl_ne_strsub Columns

Column_name	Type	Length	Nulls	Description
strsub_type	varchar2	20	0	TBD
description	varchar2	255	1	TBD
asdl	varchar2	80	1	Name of the ASDL associated with the character. If no value is provided, the substitution will be made for all ASDL commands.
csdl	varchar2	80	1	Name of the CSDL associated with the character. If no value is provided, the substitution will be made for all CSDL commands.
ne_vendor	varchar2	255	1	Name of the software vendor for the network element associated with the character. If no value is provided, the substitution will be made for all network element vendors.

Table 2-51 (Cont.) tbl_ne_strsub Columns

Column_name	Type	Length	Nulls	Description
tech_type	varchar2	255	1	Technology type for the network element associated with the character. If no value is provided, the substitution will be made for all technology types.
sftwr_load	varchar2	16	1	Software load for the network element associated with the character. If no value is provided, the substitution will be made for all software loads.
ne_str_pattern	varchar2	255	1	TBD
ne_replace_pattern	varchar2	255	1	TBD

Table indexes:

```
strsub_type, csdl, asdl, ne_vendor, tech_type, sftwr_load, ne_str_pattern,
ne_replace_pattern
```

tbl_nep

This static table, referenced by the SARM and NEP, maintains the relationship between the NEP and the secondary pool of devices which is used by the NEP to establish auxiliary connections to Host NEs.

The SARM references this table upon start up to determine the NEPs configured within the system. For each NEP, the SARM opens one or more network connections to that NEP. You can configure the number of connections.

Each NEP references this table upon start up to determine the secondary pool of devices available to all Session Managers within that NEP. It spawns a Command Processor thread for each device in this secondary pool of devices. You are responsible for populating and maintaining this table.

- **SSP_new_nep** adds a new pool of devices to this table.
- **SSP_del_nep** deletes a pool of devices from this table.
- **SSP_list_nep** lists the contents of this table.

Table 2-52 tbl_nep Columns

Column_name	Type	Length	Nulls	Description
nep_svr_cd	varchar2	8	0	The logical name of the NEP managing the secondary pool of devices. It is not the physical environment specific name listed in the interfaces file.
dialup_pool	varchar2	8	1	Secondary pool of devices available to all Session Managers within the NEP. If specified, you must define this resource pool in tbl_resource_pool. If null, there are no secondary devices available in the NEP.

Table indexes:

nep_svr_cd

tbl_nep_asdl_prog

This static table is used by the State Table Interpreter or the JInterpreter within an application server to determine the State Table program or Java class to invoke for:

- a given technology such as DMS, AXE, S12, etc.
- software load such as BCS33, PAC4, etc.
- ASDL

It is referenced by any process using the Interpreter library. It is also referenced by NEPs, SRPs, and ISPs within ASAP.

If the Interpreter executes an ASDL's State Table and is not listed in the table, it assumes that the State Table name is the same as the ASDL name. This is useful when sftwr_load and tech have no meaning, such as invoking State Tables within an SRP.

Any invocations of the CHAIN or CALL Interpreter actions require that the ASDL being chained or called has a mapping specified in this table. You are responsible for populating and maintaining this table.

- **SSP_new_asdl_map** adds a new ASDL-to-State Table mappings to tbl_nep_asdl_prog.
- **SSP_del_asdl_map** deletes ASDL-to-State Table mappings from tbl_nep_asdl_prog.
- **SSP_list_asdl_map** lists the contents of this table.

Table 2-53 tbl_nep_asdl_prog Columns

Column_name	Type	Length	Nulls	Description
tech	varchar2	16	0	The technology or type of the NE with which the Interpreter interacts, for example, DMS, S12, etc.
sftwr_load	varchar2	16	0	The Software Load of the software currently running on the NE. Since this field is non-null, you must place a value in the field even if there is no defined software load for this NE type. The table tbl_host_clli also adheres to the same convention for the software load value.
asdl_cmd	varchar2	80	0	The ASDL command passed to the Interpreter that determines the Interpreter State Table program to be loaded and executed, or passed to the JInterpreter that determines the Java program to be executed. ASAP reserves some special ASDL commands to map special requirements, such as LOGIN or CONNECTION_HANDLER, to a program.

Table 2-53 (Cont.) tbl_nep_asdl_prog Columns

Column_name	Type	Length	Nulls	Description
program	varchar2	255	0	Name of the Interpreter State Table program in tbl_nep_program to load and execute, or name of the Java connection class or Java provisioning method for JInterpreter to execute. State Tables within the NEP execute and receive the MML commands and responses. The State Tables within the SRP generally assist with the SRP work order translation process.
interpreter_type	varchar2 ['S','J']	1	0	A value of “S" indicates a State Table interpreter, whereas a value of “J" indicates a JInterpreter. A null value defaults to “S".

Table indexes:

```
tech, sftwr_load, asdl_cmd
```

tbl_nep_mux

Stores the configuration of the NEP multiplexing devices.

Table 2-54 tbl_nep_mux Columns

Column_name	Type	Length	Nulls	Description
mux	varchar2	40	0	The name of the multiplexing device. Communication parameters for this device are specified in tbl_comm_param.
host_clli	varchar2	80	0	The NE that the multiplexing device communicates with. You must first configure the NE in tbl_ne_config.
mux_dev_type	char	1	0	The type of communications protocol or interface used in the interactions between the multiplexing device type and the NE.
cp_dev_type	char	1	0	The type of communications protocol used in the interactions between the multiplexing device type and the command processor threads.

Table indexes:

```
mux
```

tbl_nep_jprogram

This table stores the mapping between a Java class name and its binary representation which is analogous to a state table name and the state table.

Table 2-55 tbl_nep_jprogram Columns

Column_name	Type	Length	Nulls	Description
program	varchar	255	0	Name of the java class in the form package.class name.
jclass	blob			The binary form of the java class.

Table indexes:

```
primary key

program
```

tbl_nep_program

This static table is used by the Interpreter within an application server and contains the State Table Program information required to perform specific processing. For example, in the NEP, interact using MML commands and responses with the appropriate NE.

Records are maintained in NPG flat files for easier editing and then compiled into their respective table records. These State Tables are called by an Interpreter in any ASAP application Server, although more frequently in NEPs and SRPs.

You are responsible for populating and maintaining the State Table program NPG files.

- **SSP_new_nep_program** adds State Table actions based on the specified program name and/or line number to tbl_nep_program.
- **SSP_del_nep_program** deletes State Table actions based on the specified program name and/or line number from tbl_nep_program.
- **SSP_list_nep_program** lists the contents of this table

Table 2-56 tbl_nep_program Columns

Column_name	Type	Length	Nulls	Description
program	varchar2	255	0	The name of the State Table Program. With this program the ASDL, Technology, and Software load map in the tbl_nep_asdl_prog static database table.
line_no	number	38	0	The line number in the State Table program which acts as a label for that instruction. The line number is similar to the ones used in BASIC programs.
action	varchar2	32	0	An action string to identify the particular action performed by the Interpreter in the Command Processor. A number of State Table actions are provided by various ASAP component libraries, for example, Interpreter, NEP, SRP, etc. You can add action functions and overwrite existing action functions as required.

Table 2-56 (Cont.) tbl_nep_program Columns

Column_name	Type	Length	Nulls	Description
act_string	varchar2	255	1	The action string associated with the State Table action. This field is required depending on the specified State Table action.
act_int	number	38	1	The action integer which, if set, represents the next line number in the State Table program. The next line number is where execution of the State Table should continue or there is a numeric field specific to the particular action function. If the action integer points to an invalid line number or is absent from the State Table program, the Interpreter registers a run time error and fails the operation. This field is required depending on the specified State Table action.

Table indexes:

```
program, line_no
```

tbl_nep_program_source

Stores State Table source code. The compiled version is saved in tbl_nep_program.

Table 2-57 tbl_nep_program_source Columns

Column_name	Type	Length	Nulls	Description
program	varchar2	255	0	The name of the State Table program. With this program the ASDL, Technology, and Software load, map in the tbl_nep_asdl_prog static database table.
line_no	number	38	0	The line number in the State Table program which acts as a label for that instruction. The line number is similar to the ones used in BASIC programs.
seq	number	38	0	Order of the source code.
source	varchar2	255	1	Line of the State Table source code.

Table indexes:

```
program, line_no, seq
```

tbl_nep_rte_asdl_nxx

ASAP can receive work orders with no Remote NE information. This static table was created to route the ASDLs to the relevant Host NE by means of a DN and ASDL command. If the Host system is unable to determine the Remote NE, the routing logic involving this table is employed in the SARM.

This table is not read into the internal memory within the SARM because such routing is rarely used in comparison to the Remote to Host NE routing mechanism. A

parameter with label MCLI and value, and SKIPCLLI must be present on the ASDL for this routing logic to be employed. You are responsible for populating and maintaining this table

SSP_new_dn_map adds new ASDL command routings by directory number to tbl_nep_rte_asdl_nxx.

SSP_del_dn_map deletes ASDL command routings from tbl_nep_rte_asdl_nxx.

SSP_list_dn_map lists the contents of this table

Table 2-58 tbl_nep_rte_asdl_nxx Columns

Column_name	Type	Length	Nulls	Description
asdl_cmd	varchar2	80	0	The ASDL command that provides ASDL specific routing capabilities by telephone number. This is important for such services as Voice Mail which is routed to separate NEs based on the ASDL.
npa	varchar2	3	0	The NPA is the first three digits in a telephone number.
nxx	varchar2	3	0	The NXX is the second three digits in a telephone number.
from_line	varchar2	4	0	The starting point of a range of telephone LINE numbers. It is the remaining four numbers in a telephone number which provide routing.
to_line	varchar2	4	0	The end point of a range of telephone LINE numbers to provide routing.
cont_typ	varchar2	1	0	Reserved for future use.
cont_nm	varchar2	8	0	Reserved for future use.
queue_nm	varchar2	64	0	The Host NE to which this ASDL is routed. The SARM determines the NEP managing this Host NE and routes the ASDL appropriately.

Table indexes:

```
asdl_cmd, npa, nxx, from_line, to_line, cont_typ
```

tbl_order_events

This table provides the order translation details for the SRT.

Table 2-59 tbl_order_events Columns

Column_name	Type	Length	Nulls	Description
translation_name	varchar2	255	0	Name of the Translation Object.
query_type	varchar2	80	0	Type of the Query.
type	char	1	0	Type of the object. • J - JMS • X - XPATH
parm	varchar2	255	1	Name of the parameter.
value	varchar2	255	1	Value for the parameter.

tbl_order_translation

This table contains the order translation script details for the SRT.

Table 2-60 tbl_order_translation Columns

Column_name	Type	Length	Nulls	Description
name	varchar2	255	0	A unique name given to the translation script as an identifier.
script_name	varchar2	255	1	Name of the script file that implements translation.
translation_type	varchar2	255	0	Type of Translation to be executed. Possible values are XSLT and DO_NOT_FORWARD.
type	char	1	0	Type of the translation object
parm	varchar2	255	1	Name of the parameter
value	varchar2	255	1	Value of the parameter
message_direction	varchar2	80	1	Translation Direction which indicates the whether this translation is for incoming orders, responses or events. Possible values are UPSTREAM, EVENT and RESPONSE.
script	BLOB	-	1	Binary form of the library/script.

tbl_resource_pool

This static table defines collections of devices which may be used by the NEP to establish connections to NEs. Such groups of devices are called resource pools.

Each NE configuration (tbl_ne_config record) determines a primary resource pool which defines one or more devices the NEP uses to connect to that NE. Such devices are not used to connect to other NEs.

Each NEP has a secondary resource pool (defined in tbl_nep) containing devices used by the NEP to establish connections to any NE managed by that NEP. Such primary and secondary resource pools are defined in this table. You are responsible for populating and maintaining this table.

- **SSP_new_resource** adds new device definitions to this table.
- **SSP_del_resource** deletes device definitions from this table.
- **SSP_list_resource** lists the contents of this table

Table 2-61 tbl_resource_pool Columns

Column_name	Type	Length	Nulls	Description
asap_sys	varchar2	8	0	This is the environment in which this database record is to be used. As the ASAP databases are environment independent, only logical representations of physical entities exist within them. • tbl_resource_pool – The table that contains specific environment-dependent information that varies between environments. • asap_sys – Distinguishes different environments. It takes the value of the environment variable, ASAP_SYS, TEST, PROD, etc., in the current environment. At run time, only those records with this field defined to be the same as the environment variable asap_sys, are loaded by the NEP.
pool	varchar2	8	0	The name of the pool of devices. It is referenced by: • tbl_nep – The secondary resource pool of the entire NEP. • tbl_ne_config – The primary resource pool of the NE.
device	varchar2	40	0	The device name. This is the name of the logical device used to establish a connection to an NE. The device corresponding to this logical device is specified by means of the communications parameter table, tbl_comm_param.
line_type	char	1	0	The communication protocol used by this device. The possible values include: • C – CORBA • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection This is enforced by the associated data rule on the datatype. For the X.25 protocol, you can use both G and M. For the X.29, you can only use G because the X.29 does not support message-based type. Such definitions are defined in the header file nep_core.h.
vs_key	number	38	-	Reserved. The shared memory segment identifier for the Virtual Screen buffer.

Table indexes:

```
Indexes

asap_sys, pool
asap_sys, device
```

tbl_srq

This dynamic table is used by the SARM and contains Service Requests (SRQs) created by the SARM from details passed by an SRP during work order translation. There is a one-to-one mapping between work orders and SRQs in the SARM. The SRQ provides a unique ID that references to the work order within ASAP.

Table 2-62 tbl_srq Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	SRQ unique ID.
srq_dd	date	-	0	Due date and time of this SRQ. This is the same as the sched_dts in the work order table.
grp_cd	char	1	0	The action of the SRQ. Possible values include: • A – ADD: addition of service. • R – REMOVE: removal of service. • C – CHANGE: change/update of existing service. • Q – QUERY: query existing service. This action field is important to the order in which SRQs are processed within the same work order and between work orders. These values are defined in asap_core.h.
srq_pri	char	1	0	Priority of this SRQ as assigned by the Host order system. It is equal to the work order priority. Possible values include: • (1) ASAP_SRQ_HIGH_PRIO – High priority SRQ. • (5) ASAP_SRQ_NORMAL_PRIO – Normal priority SRQ. • (9) ASAP_SRQ_LOW_PRIO – Low priority SRQ. These values are defined in asap_core.h. This priority field is the first field in the internal composite ASAP SRQ priority within the ASAP core followed by the SRQ due date and the SRQ action such as Remove, Change, or Add.

Table 2-62 (Cont.) tbl_srq Columns

Column_name	Type	Length	Nulls	Description
srq_stat	number	38	0	SRQ status that is updated while the SRQ is being processed by the SARM. Possible values include: • (0) HELD – SRQ is held awaiting manual intervention. • (1) INITIAL – SRQ is yet to begin provisioning. • (2) SRQ_ABORTED – SRQ has been aborted due to order update or cancel. • (3) NEP_UNAVAIL – NEP routing for the SRQ's current ASDL is temporarily unavailable. • (7) IN_PROCESSING – SRQ in process. • (12) COMPLETED – SRQ successfully completed. It may have exceptions and/or revisions. • (13) FAILED – SRQ failed. See the SRQ log details to determine the cause of failure. • (14) TRANSLATION_ERROR – The SRP could not translate the SRQ correctly but transmitted it to the SARM. • (15) SRQ_ROLLBACK – SRQ has been rolled back. • (16) REVIEW – SRQ is in Reviewed status. • (60) CMD_RETRY – An ASDL on this SRQ is currently in the ASDL Retry queue. The SARM waits for a configured period or number of retries. These definitions are defined in sarm_defs.h.
evt_dt_tm	date	-	0	Last date and time that this SRQ record was updated.
wo_id	varchar2	80	0	The ID of the work order that this SRQ belongs to.
srq_chg	char	1	1	Indicates whether the SRQ has been altered since being received by ASAP. Possible values include: • N – No revisions on the SRQ. • Y – Revisions in ASAP not to be reflected back to the Host System. • C – Revisions in ASAP to be reflected back to the Host System.

Table 2-62 (Cont.) tbl_srq Columns

Column_name	Type	Length	Nulls	Description
proc_typ	char	1	1	Type of processing required for this SRQ. Possible values include: • (I) IMMEDIATE – Immediate requests to be provisioned as they are received. • (D) DELAYED – Delayed or batch requests due in the future. Batch requests have lower priority than immediate requests. Values are defined in sarm_defs.h.
cur_csdl_seq_no	number	38	1	The sequence number of the current CSDL being processed within this SRQ. This serves as a pointer into the CSDL table which enables the SARM to determine which CSDL it had been processing. This is used by the SARM upon restarting an In Progress work order.
cur_csdl_st	number	38	1	The status of the current CSDL being processed within this SRQ. This field and the sequence number are updated by the SARM as the SRQ is being processed. Possible values of this field are detailed in the csdl_st field of the tbl_srq_csdl table.
host_clli	varchar2	80	1	The Host NE associated with the latest ASDL on the SRQ. This is updated by the SARM to the value of the current ASDL Host NE value. There may be multiple Host NEs associated with a CSDL since each ASDL may be routed to a different Host NE.

Table indexes:

```
Indexes

srq_id
wo_id
```

tbl_srq_asdl_parm

This dynamic table contains parameter name value pairs associated with a Service Request (SRQ). It allows the ASDL and rollback parameters to be defined and, where possible, provides a pointer to their location in other database tables.

Table 2-63 tbl_srq_asdl_parm Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	The SRQ ID that the parameter name value pairs are associated.

Table 2-63 (Cont.) tbl_srq_asdl_parm Columns

Column_name	Type	Length	Nulls	Description
unid	number	38	0	If the ASDL type is ROLLBACK_TYPE, (according to tbl_asdl_log) then this field is set to the value of the ASDL unique ID (asdl_unid) in the ASDL log table (tbl_asdl_log). This allows the ASDL log table to determine the rollback parameters associated with a particular ASDL. These parameter types are defined in the header file sarm_defs.h.
parm_lbl	varchar2	80	0	The parameter label, for example, LEN, or DN.
parm_vlu	varchar2	255	1	The parameter value associated with the parameter label.
vlu_hint	char	1	0	Indicates whether the parameter value is a real value or an XML document or XPath expression. Can be one of the following: • X - indicates a reference to XML document • P - indicates a reference to XPath expression • T - indicates a reference to large text (reserved for future use)

Table indexes:

```
srq_id, unid, parm_lbl
```

tbl_srq_csdl

This dynamic table contains the CSDLs for each Service Request (SRQ) listed in the SRQ table. Each SRQ can have multiple CSDLs.

Table 2-64 tbl_srq_csdl Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	Unique ID of the SRQ that is associated with the CSDL.
csdl_seq_no	number	38	0	Sequence number that distinguishes CSDLs within the SRQ.
actn_noun_lbl	varchar2	80	0	CSDL command.

Table 2-64 (Cont.) tbl_srq_csdl Columns

Column_name	Type	Length	Nulls	Description
csdl_st	number	38	0	Status of the CSDL. Possible values include: • (100) HELD_STATE – The CSDL is Held awaiting manual release within ASAP. • (101) INITIAL_STATE – The CSDL is yet to begin processing. • (102) FAILED_CSDL – The CSDL has failed. • (103) ABORTED_CSDL – The CSDL has been aborted (usually by an OCA user). • (104) COMPLETED_CSDL – The CSDL has completed successfully. • (106) ROLLBACK_COMPLETED_CSDL – The CSDL has been successfully rolled back. CSDL status values are defined in sarm_defs.h.
asdl_seq_no	number	38	0	The sequence number of the ASDL currently being processed by the SARM on the CSDL. It is updated dynamically by the SARM as the ASDLs are processed. The SARM uses this field to determine the current ASDL to be provisioned upon restart.
index_parm_cnt	number	38	0	Current index value in Indexed ASDL parameters. This field is dynamically updated by the SARM as it provisions an ASDL with a set of Indexed parameters. When the SARM restarts an In Progress order, it uses the value in this field to determine the relevant set of Indexed ASDL parameters to transmit with this ASDL. Use the getParam method to retrieve the value of the given input parameter as passed down to the JInterpreter by the SARM. Use the getIntParam method to retrieve the value of the given input parameter as passed down to the JInterpreter by the SARM and casts this value to an integer. Refer to the *ASAP Online Reference* for more information.

Table 2-64 (Cont.) tbl_srq_csdl Columns

Column_name	Type	Length	Nulls	Description
csdl_id	number	38	0	The CSDL ID in the SRP transmitted to the SARM. When the SRP transmits a CSDL to the SARM, it passes this CSDL ID which SARM stores in the field. With this, the SARM generates the csdl_seq_no field value once it receives all CSDLs from the SRP and orders them according to their respective levels. This field is maintained by the SARM so that the SRP can query the SARM for CSDL specific information. It is used by the SRP to correlate the CSDL in the SRP with the CSDL in the SARM.
asdl_route	char	1	0	Specifies the routing of the current ASDL on the CSDL. The CSDL is mapped to one or more ASDLs by the table tbl_csdl_asdl. Once the SARM acknowledges the ASDL, it looks up the appropriate entry in the tbl_asdl_config to determine the routing for this ASDL. Possible values include: • (?) TO_BE_DETERMINED – The SARM has not yet determined the routing of the first ASDL on this CSDL. • (N) ROUTE_TO_NEP – ASDL routed to the NEP. This value is defined in the header file sarm_defs.h.
csdl_type	char	1	1	The type of CSDL.
orig_seq_no	number	38	1	The original sequence number of the CSDL on the SRQ.
estimate	number	38	1	The time estimate to provision the CSDL.
start_dts	date	-	1	The date and time that the CSDL started provisioning.
abort_dts	date	-	1	The date and time that the CSDL was aborted.
failure_dts	date	-	1	The date and time that the CSDL failed provisioning.
comp_dts	date	-	1	The date and time that the CSDL completed provisioning.
update_dts	date	-	1	The date and time that the CSDL was last updated.
update_uid	varchar2	64	1	The user who last updated the CSDL.
prov_sequence	number	38	1	The provisioning sequence of the CSDL on the SRQ. You can specify an alternate provisioning sequence to the one originally received from the originating system.

Table indexes:

```
srq_id, csdl_seq_no
```

tbl_srq_log

This dynamic table contains information logged for each SRQ. The log is a history of events that occurred on each SRQ, including Switch History information of the NE responses. If the Switch History is greater than 195 characters, the entry splits into two entries.

Through an API, the SARM and NEP write to this database table during the provisioning process. The SRP queries this table through an API to retrieve selected records for a particular work order.

You can view audit log information through the OCA Client. For more information, refer to the *ASAP OCA User Guide*.

Table 2-65 tbl_srq_log Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	ID of the Service Request.
srq_log_identity	number	20	0	Unique ID of the log message in this table. This is an identity field automatically generated by the RDBMS upon insertion. This replaces the earlier field for performance reasons.
evt_dt_tm	date	-	0	Date and time of the logged event.
csdl_seq_no	number	38	0	Sequence number of the CSDL within the SRQ that the event is associated with. This is the same as the csdl_seq_no field in the CSDL table.
srq_stat	number	38	0	Status of the SRQ at the time the event occurred. For possible values, see srq_stat in tbl_srq, "tbl_srq."
csdl_st	number	38	0	Status of the CSDL when the event occurred. For possible values, see csdl_st in tbl_srq_csdl, "tbl_srq_csdl."

Table 2-65 (Cont.) tbl_srq_log Columns

Column_name	Type	Length	Nulls	Description
srq_evt	varchar2	8	0	The SRQ Log Event. The SRP may inquire for the SRQ log by specifying particular SRQ events of interest in the inquiry RPCs. Possible values include: • SRQ_INFO_EVENT "INFO" – Information messages. • SRQ_ERROR_EVENT "ERROR" – Error messages. • NE_CMD_EVENT "NE_CMD" – Command entered to the NE. • NE_RESP_EVENT "NE_RESP" – Response from the NE. These values are defined in the header file sarm_defs.h. You must set WO_AUDIT_LEVEL in ASAP.cfg to 2 to generate SRQ_ERROR_EVENTS.
evt_text	varchar2	255	1	Text description of the event. For NE responses, it contains 255 characters at a time (including newline characters) from the NE generated report. This should be considered when the information is being displayed by a front-end user interface. This allows each log record to contain more NE response information than if one NE response record was contained in each SRQ log record. If generated by an application, this text describes the event as displayed to you. All control characters x0 to x1F, except x9, xA, and xD (tab, cr and nl), are stripped from the event text after it is retrieved from the database and before used in the XML document. This behavior is in compliance with XML 1.0 specification.
asdl_unid	number	38	1	The ID of the ASDL that generated the log entry. If there is no current ASDL for the log entry, set this field to null.

Table indexes:

```
srq_id, srq_log_identity
```

tbl_srq_parm

This dynamic table contains global and CSDL name value pairs associated with an SRQ. It allows different types of parameters to be defined and where possible, provides a pointer to their location in other database tables. There is an index on this table which allows efficient query access to the global and CSDL parameters.

Table 2-66 tbl_srq_parm Columns

Column_name	Type	Length	Nulls	Description
srq_id	number	38	0	The SRQ ID that is associated with the parameter name value pairs.
parm_typ	char	1	0	The type of SRQ parameter. Possible values include: • (P) GLOBAL_TYPE – Parameters are set by one CSDL in an SRQ and referenced by others (they are global within the SRQ). • (C) CSDL_TYPE – Parameters are local to the current CSDL being processed within the SRQ. They are referenced by other ASDLs within the same CSDL. These parameter types are defined in the header file sarm_defs.h.
unid	number	38	0	If the parameter type is CSDL_TYPE, this field equals the CSDL sequence number (csdl_seq_no) in the CSDL tbl_srq_csdl. This field is not used if the parameter type is GLOBAL_TYPE.
parm_lbl	varchar2	80	0	The parameter label, for example, LEN or DN.
parm_vlu	varchar2	255	1	The parameter value associated with the parameter label.
parm_subvlu	varchar2	80	1	Contains the first 32 bytes of the global or CSDL parameter value.
vlu_hint	char	1	0	Indicates whether the parameter value is a real value or an XML document or XPath expression. Can be one of the following: • X - indicates a reference to XML document • P - indicates a reference to XPath expression • T - indicates a reference to large text (reserved for future use)

Table indexes:

```
Indexes

srq_id, parm_typ, unid, parm_lbl
parm_subvlu, parm_lbl
```

tbl_srt_bundle

This table contains SRT service bundles and/or its spawning details for the SRT.

Table 2-67 tbl_srt_bundle Columns

Column_name	Type	Length	Nulls	Description
service_id	varchar2	80	0	Service Identifier.
description	varchar2	1024	1	Description of the bundle or service action.
spawn_parm	varchar2	255	1	Name of the spawning parameter on the order that will cause this service bundle or service action to be added to the order if it has the correct value
spawn_value	varchar2	255	1	Value of the spawning parameter that will cause the service bundle or service action to be added to the order.
service_type	char	1	0	The service_type field indicates whether the table entry is a service bundle or a service action. • B – Service bundle • C – Service action

tbl_srt_bundle_csdl

This table contains the service action spawning information for the SRT.

Table 2-68 tbl_srt_bundle_csdl Columns

Column_name	Type	Length	Nulls	Description
service_id	varchar2	80	0	Service Identifier.
csdl_cmd	varchar2	80	0	Name of the CSDL.
csdl_seq_no	number	20	0	Sequence number of the CSDL.
cond_flag	char	1	0	Conditional flag for the CSDL. Possible values are: • A - ALWAYS • E - EQUALS • D - DEFINED • N - NOT_DEFINED
label	varchar2	80	1	The parameter label.
value	varchar2	255	1	Value of the parameter.
eval_exp	varchar2	255	1	Contains combination of parameter names, operators, and values to which the parameters are compared.
inc_ord_resp	char	1	1	Identifies whether order data need to be included in the response or not. • Y – Yes • N – No
description	varchar2	1024	1	Description of the service bundle or service action.

tbl_srt_config_reload

This table specifies the SRT configuration reload time.

Table 2-69 tbl_srt_config_reload Columns

Column_name	Type	Length	Nulls	Description
load_dts	date	-	0	Date and time at which the SRT configuration was reloaded.

tbl_srt_correlation

This table provides the correlation details for the SRT work order.

Table 2-70 tbl_srt_correlation Columns

Column_name	Type	Length	Nulls	Description
correlation_id	varchar2	80	0	SRT Correlation identifier
asap_id	varchar2	80	0	ASAP Work Order identifier

tbl_srt_csdl_parm

This table provides the service action parameter mapping details for the service bundle.

Table 2-71 tbl_srt_csdl_parm Columns

Column_name	Type	Length	Nulls	Description
service_id	varchar2	80	0	Service Identifier.
csdl_cmd	varchar2	80	0	The CSDL command.
csdl_seq_no	number	20	0	Sequence number of the CSDL.
parm_seq_no	number	20	0	Sequence number of the parameter.
bundle_label	varchar2	80	0	The upstream label used in transmitting the parameters for provisioning.
csdl_label	varchar2	80	0	The CSDL label used in transmitting the parameters for provisioning.
default value	varchar2	255	1	Default Value.
parm_type	char	1	0	Type of the parameter. Possible values include: • R - Required Scalar • O - Optional Scalar • C - Required Compound • N - Optional Compound • M - Required Indexed • I - Optional Indexed

tbl_srt_ctx

This table provides the correlation details for the SRT work order.

Table 2-72 tbl_srt_ctx Columns

Column_name	Type	Length	Nulls	Description
asap_id	varchar2	80	0	ASAP identifier.
name	varchar2	1024	0	Name of the parameter.
value	varchar2	1024	0	Value of the parameter.
type	varchar2	1024	0	Type of the parameter.

srt_header_mapping

This table enumerates XPath names and values to allow additional event XML message body data to be placed in the JMS header properties. This supports the inclusion of extended event data in JMS headers without additional database queries. If there are no entries in this table, then the default behavior occurs for JMS header creation. See "tbl_event_dataset" and "tbl_event_template."

The table contains of XPath names and values. The SRT iterates through the table entries and runs the configured XPaths. The returned name/value pairs are added to the JMS header properties.

If the XPath in the configuration name attribute returns multiple results, all returned values are added to the header. The values of these parameters are the result list of the value attribute. The name/value pairs are paired in the order they were returned by the XPath functions.

If the parameter name XPath returns more results than the value XPath the remaining values will be left blank.

If the parameter name XPath returns fewer results than the value XPath results, extra value XPath results are ignored.

Note: An XPath within single quotes represents a constant.

Table 2-73 srt_header_mapping Columns

Column_name	Type	Length	Nulls	Description
xpath_name	varchar2	1024	0	xpath to generate a name.
xpath_value	varchar2	1024	0	xpath to generate a value

Examples:

To return a specific information parameter on all event types, create a record with values similar to the following:

Name – 'IMSI'

Value – /*[name()=mslv-sa:completeEvent]//extendedWoProperties/
extendedWoProperty[name='IMSI']/value

To return all extendedWoProperties on a completed event, create a record with values similar to the following:

Name – /*[name()=mslv-sa:completeEvent]//extendedWoProperties//name

Value – /*[name()=mslv-sa:completeEvent]//extendedWoProperties//value

tbl_srt_lookup

This table provides lookup details used by the SRT.

Table 2-74 tbl_srt_lookup Columns

Column_name	Type	Length	Nulls	Description
name	varchar2	255	0	Name of the lookup.
type	varchar2	128	0	Type of the lookup.
cache_scope	varchar2	32	1	Scope of the lookup. Scope can be either: NONE, NODE or SYSTEM.
cache_timeout	varchar2	20	1	The amount of time in milliseconds to cache a value if the scope is SYSTEM.
cache_max_size	number	20	1	The maximum number of entries to cache if the scope is SYSTEM. Once MaxSize is reached, least recently used values will be dropped.

tbl_srt_lookup_input

This table provides lookup input details used by the SRT.

Table 2-75 tbl_srt_lookup_input Columns

Column_name	Type	Length	Nulls	Description
lookup_name	varchar2	255	0	Name of the lookup
parm_name	varchar2	255	0	Name of the input parameter
parm_type	char	1	0	Type of the parameter. Possible values are: • V - represents Adapter Properties Value • L - represents Lookup
parm_value	varchar2	255	0	Value of the parameter associated with the parameter name. If parm_type is L then parm_value contains an XPath, otherwise parm_value contains an instance string value.
parm_source	varchar2	255	0	Indicates the source XML document for the order data. Possible values are: • if parm_type is L, then parm_source contains the name of a data provider / Lookup • otherwise, if parm_source is ASAP_SRT_ORDER, this means that the source XML document contains the order data

tbl_srt_lookup_output

This table provides lookup output details used by the SRT.

Table 2-76 tbl_srt_lookup_output Columns

Column_name	Type	Length	Nulls	Description
lookup_name	varchar2	255	0	Name of the lookup.
parm_name	varchar2	255	0	Name of the output parameter.
parm_type	char	1	0	Type of the parameter. Possible values is: • X - represents XPATH
parm_value	varchar2	255	0	Value of the parameter.

tbl_srt_query_spawn

This table provides query spawning details for the SRT.

Table 2-77 tbl_srt_query_spawn Columns

Column_name	Type	Length	Nulls	Description
parm_name	varchar2	255	0	Name of the parameter
req_exp	varchar2	1024	0	Regular expression used by the SRT.
eval_cond	char	1	0	Evaluation condition

tbl_stubs

This dynamic table is used in the ASAP High Availability configuration to maintain a copy of the critical data relating to any work order in either ASAP site. As the SARM processes a work order locally, the other SARM is notified of any critical updates performed on the order by means of SARM to SARM communication. The other SARM updates its stub table with the critical update information.

Table 2-78 tbl_stubs Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The Work order ID
wo_stat	number	38	0	The status of the work order. Refer wo_stat in "tbl_wrk_ord (user-created database table)" for a list of possible status values.
wo_cmd	number	38	0	The work order command.
srp_id	varchar2	8	0	The SRP from which this work order originated.
sched_dts	date	-	0	The due date and time of the work order.
crit_seq_no	number	38	1	The critical sequence number representing the current critical update that was last applied to the work order.
parent_wo	varchar2	80	1	The parent work order of this order.

Table indexes:

```
Wo_id, parent_wo
```

tbl_test_rpc_parm

This table provides the details about the RPC parameters used by SRT.

Table 2-79 tbl_test_rpc_parm Columns

Column_name	Type	Length	Nulls	Description
rpc	varchar2	80	0	The Work order ID
seq_no	number	38	0	The status of the work order. Refer wo_stat in "tbl_wrk_ord (user-created database table)" for a list of possible status values.
parm_lbl	varchar2	80	0	The work order command.
parm_typ	char	1	0	The SRP from which this work order originated
default_vlu	varchar2	255	1	The due date and time of the work order

Table indexes:

```
rpc, seq_no
```

tbl_stat_text

Stores labels for the OCA client.

- **SSP_new_stat_text** adds new static text labels to tbl_stat_text.
- **SSP_del_stat_text** deletes static text labels from tbl_stat_text.
- **SSP_list_stat_text** lists the contents of this table.

Table 2-80 tbl_stat_text Columns

Column_name	Type	Length	Nulls	Description
stat_id	varchar2	10	0	ID for a group of labels.
status	number	38	1	Integer key field for grouping.
code	varchar2	20	1	String key field for grouping.
stat_text	varchar2	100	0	Text describing the label.

Table indexes:

```
Indexes

stat_id, status
stat_id, code
```

tbl_unload_sp

This table provides information about the stored procedures used to insert/delete/ data to/ from the ASAP tables. This table will be used by the utilities to load/unload the data in the ASAP tables.

This table is available in both the Control and SARM databases, containing data appropriate to the respective database.

Table 2-81 tbl_unload_sp Columns

Column_name	Type	Length	Nulls	Description
seq_no	number	38	0	Sequence number of the table.
tbl_name	varchar2	40	0	Name of the ASAP table.
new_sp	varchar2	40	1	Stored procedure used to insert the data into the table.
del_sp	varchar2	40	1	Stored procedure used to delete the data from the table.
list_sp	varchar2	40	1	Stored procedure used to list the data in the table.

Table indexes:

```
seq_no
```

tbl_unload_param

This table is used by utilities. It provides information about the parameters inserted/deleted by stored procedures.

This table is available in both the Control and SARM databases, containing data appropriate to the respective database.

Table 2-82 tbl_unload_param Columns

Column_name	Type	Length	Nulls	Description
seq_no	number	38	0	Sequence number of the table.
col_number	number	38	0	Column number of the parameter in the table.
para_name	varchar2	80	0	Name of the Parameter.
default_flag	number	38	0	Default Flag.
sp_type	char	3	0	Type of the stored procedure.
rows_int	number	38	0	Number of rows.

Table indexes:

```
seq_no, para_name, sp_type
```

temp_wrk_ord

This dynamic table details the supplemental work order information required by the ASAP functionality. The columns are similar to tbl_wrk_ord table. For details, refer to "tbl_wrk_ord (user-created database table)."

Table 2-83 temp_wrk_ord Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The work order ID that uniquely identifies the work order in the ASAP core.
sched_dts	Date	-	0	The scheduled date and time for provisioning to occur on work orders due in the future. For immediate work orders, set it to the current time.
wo_stat	number	38	0	The status of the work order. It is updated by the SARM as the work order is being processed. For possible values, refer to the "tbl_wrk_ord (user-created database table)" table.
comp_dts	Date	-	1	The completion date of all provisioning associated with the work order in ASAP.
srp_id	varchar2	8	0	The logical name of the front-end SRP that notifications and results are sent to for a particular work order.
update_dts	Date	-	0	The date and time of the last update on the work order within the ASAP core.
org_unit	varchar2	8	0	The Organization Unit of the person or group to whom notification is sent should particular events occur on the work order.
orig_login	varchar2	64	1	The original login ID of the user who initiated the work order in the Host order system. This information is used for display, notification, and diagnostic purposes.
revs_flag	char	1	0	The revisions flag on the work order to indicate if the work order was revised by the OCA client.
exceptions	char	1	0	Exceptions flag indicating to the SRP if there are any exceptions in the completion of the work order. For more details, refer to the "tbl_wrk_ord (user-created database table)" table.
pend_cancel	char	1	1	This flag indicates whether there is a pending order cancellation for this work order.
rollback_stat	number	38	0	The rollback status of the work order.
command	number	38	0	This field is transmitted by the SRP to the SARM and informs the SARM processing command to apply it to the work order.

Table 2-83 (Cont.) temp_wrk_ord Columns

Column_name	Type	Length	Nulls	Description
crit_seq_no	number	38	1	The sequence number of the last critical update performed on this work order. This is updated each time a critical update to the work order has been performed.
lock_uid	varchar2	64	1	The user who last locked the work order for updating.
lock_dts	Date	-	1	The date and time that the work order was last locked for updating.
start_dts	Date	-	1	The date and time that the work order started provisioning.
asdl_timeout	number	38	1	If set, the ASDL time-out interval to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter ASDL_TIMEOUTS.
parent_wo	varchar2	80	1	If set, the parent work order on which the work order is dependent. The parent order must be completed for the work order to begin provisioning.
wo_timeout	number	38	1	If set, the work order time-out interval to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter ORDER_TIMEOUT.
asdl_retry_num	number	38	1	If set, the number of ASDL retries to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter NUM_TIMES_RETRY.
asdl_retry_int	number	38	1	If set, the time period in seconds between ASDL retries to be used on the work order instead of the system-wide SARM default.
wo_rback	char	1	1	If set to: • (Y)es – A flag specifies whether to explicitly roll back the order in the event of failure. • (N)o – The order is not rolled back. • (D)efault – The SARM receives the setting from the SRP.
stub_update_req	char	1	1	Indicates whether Stub update is required or not.
asdl_delay_fail	char	1	1	If set, a flag which specifies whether or not to treat hard errors encountered on the work order as delayed failures, therefore allowing the work order to finish processing before failing.

Table 2-83 (Cont.) temp_wrk_ord Columns

Column_name	Type	Length	Nulls	Description
max_delay_fail	number	38	1	If set, the number of ASDL delayed failures to allow before failing a batch work order. This allows the overriding of the system-wide SARM default which is specified by the configuration parameter, BATCH_THRESHOLD.
srq_id	number	38	1	The ID of the latest SRQ associated with the work order. There is a single SRQ related to a given work order.
is_future_dated	char	1	1	If set to: • Y – It indicates that the work order is future-dated. • N – The work order is not future-dated. This is used by the ADMIN Server.
batch_group	varchar2	80	1	The batch group ID of the work order.
extsys_di	varchar2	128	1	An external system ID that is passed with the work order. It is not used by the SARM, but is used by the SRP for proper routing to upstream systems.
rollback_exceptions	char	1	0	The rollback exceptions flag returned by the SARM on the work order rollback completion notification. Indicates whether there are exceptions (i.e. ASDL failure) during rollback of a work order.
point_of_no_return	number	38	0	Values are: • -1 – No rollback if work order fails • 0 – (default). Normal rollback behavior. No 'point of no return' functionality. • >0 – ASDL_SEQ_NO. This ASDL is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this ASDL but no further.
failure_reason	varchar2	255	1	Provides the failure reason for a work order that fails during provisioning.

Table indexes:

```
wo_id, srq_id
```

tbl_uid_pwd

If the SARM configuration variable SECURITY_CHECK is enabled, this static table is referenced to authorize access to the SARM from the SRP as part of the SRP to SARM protocol.

For the security check to be validated, the user ID and password on the work order must be defined in the table with an active status. If a security violation is detected, the SARM rejects the work order with a security violation, ASAP_STAT_SECURITY_VIOLATION (103).

This provides a central security mechanism to ensure that only properly validated work orders are received by the SARM from all SRPs in the system.

- **SSP_new_userid** adds a new user account for the SARM to control access from the SRP in tbl__uid_pwd.
- **SSP_del_userid** deletes a user account for the SARM to control access from the SRP in tbl__uid_pwd.
- **SSP_list_userid** lists the contents of this table.

Table 2-84 tbl_uid_pwd Columns

Column_name	Type	Length	Nulls	Description
userid	varchar2	64	0	The user ID for the security check.
pwd	varchar2	30	0	The associated password.
status	varchar2	40	1	Your current status. If set to ACTIVE, then you can access. If not, access is denied.

Table indexes:

```
userid
```

tbl_unid

This dynamic table manages unique IDs required by other tables. It is present in most user-created databases and provides a method of generating a serial field.

Table 2-85 tbl_unid Columns

Column_name	Type	Length	Nulls	Description
unid_type	varchar2	8	0	A unique code identifying the UNID type. This allows there to be many different UNID values for different types of UNID.
unid	number	38	0	The UNID value for a particular type.
pad1	char	255	0	Padding to make a table row occupy a page. This reduces concurrence on the database data page by different database processes.
pad2	char	255	0	Same as pad1.
pad3	char	255	0	Same as pad1.
pad4	char	255	0	Same as pad1.

Table indexes:

```
Unique

unid_type
```

tbl_user_err

This static table provides a mechanism to define user exit codes and map them to one of the base ASDL exit types. For more information on user exit types, see the *ASAP Cartridge Development Guide*.

If the State table returns a user-defined exit type, the NEP checks whether there is a base exit type defined for the specified ASDL and exit type. If so, then the base exit type is used. If not, the NEP determines whether there is a user-defined error type associated with the user-defined exit type.

Some initial data is provided as part of the core system. You are responsible for populating and maintaining this table.

Pattern matching is ordered by length (search_pattern), csdl, asdl, ne_vendor, sftwr_load and tech_type. Pattern matching starts from the most specific specification to the least specific. The most specific specification is one that has all columns within the table filled.

- **SSP_new_err_type** adds a new mapping of user-defined error types.
- **SSP_del_err_type** deletes the mapping of user-defined error types.
- **SSP_list_err_type** list the contents of this table

Table 2-86 tbl_user_err Columns

Column_name	Type	Length	Nulls	Description
csdl	varchar2	80	1	The CSDL that is executing. Error types can be defined for user_type and CSDL combinations.
asdl	varchar2	80	1	The ASDL that is executing. Error types can be defined for user_type and ASDL combinations.
vendor	varchar2	255	1	The vendor of the network element.
tech_type	varchar2	255	1	The technology of the network element.
sftwr_load	varchar2	16	1	Software version of the host network element.
search_pattern	varchar2	255	1	Regular expression pattern that is used to match on network element responses.
user_type	varchar2	20	0	User-defined ASDL exit type.

Table 2-86 (Cont.) tbl_user_err Columns

Column_name	Type	Length	Nulls	Description
base_type	varchar2	20	0	The base ASDL exit type where this user specified ASDL exit type maps to. The base types include: • SUCCEED – ASDL executed successfully. • FAIL – ASDL encountered a hard error. • RETRY – ASDL to be retried in future. • MAINTENANCE – ASDL failed because the NE is currently unavailable to receive provisioning requests. • SOFT_FAIL – ASDL generates an error occurs that should not halt the processing of the order. • DELAYED_FAIL – ASDL failed during provisioning. The SARM skips any subsequent ASDL in the CSDL, continues provisioning at the next CSDL, and then fails the order. • STOP – ASDL is stopped. Refer to the *ASAP Cartridge Development Guide* for more detailed descriptions of these base_types.
description	varchar2	255	1	Description of the user exit type.

Table indexes:

```
Unique

csdl, asdl, ne_vendor, tech_type, sftwr_load, search_pattern, user_type
```

tbl_user_err_threshold

With this static table you can specify thresholds for a specific user exit from the Interpreter State Table by Host NE and ASDL command.

The SARM maintains three counters for each Host NE, ASDL, and user exit type. If a counter exceeds its user configured threshold, the SARM issues the appropriate event, if defined. In the Control database you can configure any system events generated from this table to map to relevant system alarms as required.

- **SSP_new_user_err_threshold** adds a new user-defined error threshold or set of thresholds to tbl_user_err_threshold.
- **SSP_del_user_err_threshold** deletes a user-defined error threshold or set of thresholds from tbl_user_err_threshold.
- **SSP_list_user_err_threshold** lists the contents of this table

Table 2-87 tbl_user_err_threshold Columns

Column_name	Type	Length	Nulls	Description
host_clli	varchar2	80	0	The Host NE.
asdl_cmd	varchar2	80	0	The ASDL command.
user_type	varchar2	20	0	The user State Table exit type.
minor_threshold	number	4	1	The number of user exits for this Host NE and ASDL before the minor event is generated.
minor_event	varchar2	8	1	The minor system event to be generated.
major_threshold	number	4	1	The number of user exits for this Host NE and ASDL before the major event is triggered.
major_event	varchar2	8	1	The major system event generated.
critical_threshold	number	4	1	The number of user exits for this Host NE and ASDL before the critical event is triggered.
critical_event	varchar2	8	1	The critical system event generated.

Table indexes:

```
Unique

host_clli, asdl_cmd, user_type
```

tbl_usr_wo_prop

This table is used to dynamically configure work order properties. It consists of WO ID and name/value pair for a user-defined work order properties. This table supports only string type user-defined work order properties.

Table 2-88 tbl_usr_wo_prop Columns

Column Name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The ID of the work order.
name	varchar2	80	0	The name of the work order property.
value	varchar2	255	1	The value given to the work order property.
value_hint	char	1	0	Indicates whether the parameter value is a real value (XML document) or a reference ID (XPATH expression). Can be one of the following: • X - indicates a reference to XML document • P - indicates a reference to XPath expression • T - indicates a reference to large text (reserved for future use)

Table indexes:

```
Unique

wo_id, name
```

tbl_wo_audit

This dynamic table tracks the work order status. It is populated based on the wo_audit column in "tbl_msg_convert" and the WO_AUDIT_LEVEL parameter in ActivationConfig.xsd and ASAP.cfg.

You can view audit log information through the OCA Client. For more information, refer to the *ASAP OCA User Guide*.

Table 2-89 tbl_wo_audit Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The ID of the work order.
wo_audit_log_identity	number	20	0	Unique ID of the audit message. This is an identity field automatically generated by the RDBMS upon insertion.

Table 2-89 (Cont.) tbl_wo_audit Columns

Column_name	Type	Length	Nulls	Description
wo_stat	smallint	1	0	The status of the work order internally within ASAP. It is updated by the SARM as the work order is being processed. Possible values for this field include: • (101) WO_LOADING – The work order is being loaded into the SARM from the SRP. • (102) WO_INIT – The work order is in the Initial state awaiting provisioning. • (103) WO_IN_PROGRESS – The work order is currently being provisioned. • (104) WO_COMPLETE – The work order has been completed. • (200) WO_GET_STATUS – Transient state within the SARM. • (221) WO_STOP_WAIT – The work order has been stopped and is being rolled back. • (222) WO_STOPPED – The work order is currently stopped. • (246) WO_REVIEW – The work order is in a Review state (similar to held). • (247) WO_CANCEL_WAIT – The work order has a cancellation request awaiting. • (249) WO_LOCK – The work order is in a Locked state. • (250) WO_ABORT – The work order has been aborted. • (251) WO_TIME_OUT – Work order processing has exceeded the time-out interval. • (252) WO_CANCELLED – The work order has been cancelled, usually from the Host system. • (253) WO_FAILED – The work order provisioning has failed. • (254) WO_HELD – The work order is placed in a Held state. • (255) WO_TRANSLATION_FAIL – The work order translation failed in the SRP, but was transmitted to the SARM to facilitate manual intervention. These values are defined in the header file sarm_defs.h

Table 2-89 (Cont.) tbl_wo_audit Columns

Column_name	Type	Length	Nulls	Description
srq_evt	varchar2	8	1	The SRQ Log Event. The SRP may inquire for the SRQ log by specifying particular SRQ events of interest in the inquiry RPCs. Possible values include: • SRQ_INFO_EVENT "INFO" – Information messages. • SRQ_ERROR_EVENT "ERROR" – Error messages. • NE_CMD_EVENT "NE_CMD" – Command entered to NE. • NE_RESP_EVENT "NE_RESP" – Response from NE. These values are defined in the header file sarm_defs.h.
evt_dt_tm	date	-	1	Last date and time the SRQ record was updated. This value is referenced when performing audit log queries in the OCA client.
evt_text	varchar2	255	1	Description of the event.
user_id	varchar2	64	1	The user ID.
sched_dts	date	-	1	The scheduled date and time for provisioning to occur on this work order. It is used for work orders that are due in the future. For immediate work orders, set it to the current time. This value is referenced when performing work order queries in the OCA client.
priority	varchar2	21	0	Priority of the SRQ as assigned by the Host order system. It is equal to the work order priority. Possible values include: • (1) ASAP_SRQ_HIGH_PRIO – High priority SRQ. • (5) ASAP_SRQ_NORMAL_PRIO – Normal priority SRQ. • (9) ASAP_SRQ_LOW_PRIO – Low priority SRQ. These values are defined in asap_core.h. This priority field is the first field in the internal composite ASAP SRQ priority within the ASAP core followed by the SRQ due date and the SRQ action such as Remove, Change, or Add.
batch_group	varchar2	80	1	The batch group ID of the work order.
parent_wo	varchar2	80	1	If set, the parent work order on which this order is dependent. The parent work order must be completed for provisioning to begin.

Table 2-89 (Cont.) tbl_wo_audit Columns

Column_name	Type	Length	Nulls	Description
org_unit	varchar2	8	0	The Organization Unit of the person or group to whom notification is sent should particular events occur on the work order. It is used for notification purposes and user group determination.
grp_cd	char	1	0	The action of the SRQ. Possible values include: • (A) ADD – Addition of service. • (R) REMOVE – Removal of service. • (C) CHANGE – Change/update of existing service. • (Q) QUERY – Query existing service. This field determines the order in which SRQs are processed within the same work order and SRQs between work orders. These values are defined in asap_core.h.
wo_event_time	date	-	1	The date and time that the work order event was placed into the audit log.
wo_event_location	varchar2	255	1	The location of the work order event. For example, which pending queue the ASDL was placed in.

Table indexes:

```
wo_id, wo_audit_log_identity
```

tbl_wo_event_queue

This dynamic table stores a back-up copy of all SRP events generated. Completed events are purged on a periodic basis.

Table 2-90 tbl_wo_event_queue Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The work order ID.
event_unid	number	4	0	A unique ID of this event. If an alarm is generated by this event, an alarm log entry in tbl_alarm_log is created with this event unid value.
event_type	number	4	0	The event type. It specifies if a system alarm is generated by accessing the static table, tbl_event_type.
event_status	number	4	0	The current status of the event.
srp_id	varchar2	8	0	The logical name of the front end SRP where notifications and results are sent for this particular work order, for example, SRP_EMUL.
event_dts	date	-	0	The date and time of the system event.

Table 2-90 (Cont.) tbl_wo_event_queue Columns

Column_name	Type	Length	Nulls	Description
start_dts	date	-	0	The date and time of when provisioning starts for the ASDL. The difference between this value and queue_dts represents the time the ASDL spent in the queue of pending ASDLs to that Host NE before being transmitted to the NEP for provisioning.
complete_dts	date	-	0	The date and time of when the ASDL provisioning is completed.
estimate	number	4	1	The work order processing estimate returned to the SRP by the SARM after the work order was transmitted to the SARM by the SRP.
misc	varchar2	80	1	This field specifies miscellaneous information received by the SARM through the asap_wo_begin RPC. It contains another work order identifier associated with the primary work order.
rev_flag	char	1	1	The revisions flag returned by the SARM on the work order completion notification.
exceptions	char	1	1	The exceptions flag returned by the SARM on the work order completion notification.
mach_clli	varchar2	80	1	Remote NE.
host_clli	varchar2	80	1	Host NE to which the Remote NE is connected.
reason	varchar2	80	1	A description of the system event.
csdl_seq_no	number	4	1	Sequence number of the CSDL within the logical work order.
csdl_id	number	4	1	The ID of the CSDL being processed when the Information parameter is generated. This CSDL ID is the same as the ID in tbl_srq_csdl. It allows the SRP to track the CSDL that this information parameter belongs to.
timeout_status	number	4	1	The status of the work order when a timeout occurs. Possible values include: • (90) ASAP_TIMEOUT_EXECUTING – The work order timed out but was still executing. • (91) ASAP_TIMEOUT_FAIL – The work order timed out and failed.
queue_tm	float	8	1	Timestamp when the event was generated.

Table 2-90 (Cont.) tbl_wo_event_queue Columns

Column_name	Type	Length	Nulls	Description
xaction_type	varchar2	16	1	The type of SRP event to be sent. Possible values include: • WO_ACCEPT • WO_ESTIMATE • WO_STARTUP • WO_COMPLETION • WO_FAILURE • WO_SOFT_ERROR • WO_BLOCKED • WO_ROLLBACK • WO_TIMEOUT • NE_UNKNOWN • NE_UNAVAILABLE • NE_AVAILABLE
old_wo_stat	number	4	1	The previous status of the work order in the SARM.
new_wo_stat	number	4	1	Status of the new work order.
status	number	4	1	The current status of the work order in the SARM.
extsys_id	varchar2	128	1	An external system ID passed with the work order. It is not used by the SARM, but is used by the SRP for proper routing to upstream systems.
rollback_exceptions	char	1	1	The rollback exceptions flag returned by the SARM on the work order rollback completion notification.

Table indexes:

```
Indexes

wo_id, event_unid
event_status, srp_id
```

tbl_wrk_ord (SARM)

This dynamic table details the essential work order information required by the ASAP core functionality. It contains details required by the provisioning process but does not contain customer-specific work order details.

Table 2-91 tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The work order ID that uniquely identifies the work order in the ASAP core. This ID is the same as the one used in the Host system and allows easier user reference to orders in the OCA client.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
sched_dts	date	-	0	The scheduled date and time for provisioning to occur on work orders due in the future. For immediate work orders, set it to the current time.
wo_stat	number	1	0	The status of the work order. It is updated by the SARM as the work order is being processed. Possible values include: • (101) WO_LOADING – The work order is being loaded into the SARM from the SRP. • (102) WO_INIT – The work order is in the Initial state awaiting provisioning. • (103) WO_IN_PROGRESS – The work order is being provisioned. • (104) WO_COMPLETE – The work order is completed. • (200) WO_GET_STATUS – Transient state within the SARM. • (221) WO_STOP_WAIT – The work order has stopped and is being rolled back. • (222) WO_STOPPED – The work order is stopped. • (246) WO_REVIEW – The work order is in a Reviewed state, similar to Held. • (247) WO_CANCEL_WAIT – The work order has a cancellation request waiting. • (249) WO_LOCK – The work order is in a Locked state. • (250) WO_ABORT – The work order has been aborted. • (251) WO_TIME_OUT – Work order processing has exceeded the timeout interval. • (252) WO_CANCELLED – The work order has been cancelled, usually from the Host system. • (253) WO_FAILED – The work order provisioning has failed. • (254) WO_HELD – The work order is placed in a Held state. • (255) WO_TRANSLATION_FAIL – The work order translation failed in the SRP, but was transmitted to the SARM to facilitate manual intervention. These values are defined in the header file sarm_defs.h.
comp_dts	date	-	1	The completion date of all provisioning associated with the work order in ASAP.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
srp_id	varchar2	8	0	The logical name of the front-end SRP that notifications and results are sent to for a particular work order.
update_dts	date	-	0	The date and time of the last update on the work order within the ASAP core.
org_unit	varchar2	8	0	The Organization Unit of the person or group to whom notification is sent should particular events occur on the work order.
orig_login	varchar2	64	1	The original login ID of the user who initiated the work order in the Host order system. This information is used for display, notification, and diagnostic purposes.
revs_flag	char	1	0	The revisions flag on the work order to indicate if the work order was revised by the OCA client. The possible values include: • (Y) ASAP_WO_REVISIONS – The work order has associated revisions. • (N) ASAP_WO_NO_REVISIONS – The work order has no associated revisions. This flag is passed back to the SRP by the SARM in the Work Order Completion Event. Such values are defined in the header file asap_core.h.
exceptions	char	1	0	Exceptions flag indicating to the SRP if there are any exceptions in the completion of the work order. Such exceptions are generally the result of a “Fail but Continue" status being returned to the SARM for one of the ASDLs on the work order. This field is set by the SARM and communicated to the relevant SRP, which then requests the exception details. The possible values include: • (Y) ASAP_WO_EXCEPTIONS – The work order completed with exceptions. • (N) ASAP_WO_NO_EXCEPTIONS – The work order completed without any exceptions. The flag is passed back to the SRP by the SARM in the Work Order Completion Event. The values are defined in the header file asap_core.h.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
pend_cancel	char	1	1	This flag indicates if there is a pending order cancellation for this work order. Upon completion of the next ASDL on the order, the SARM checks this flag. If set, the SARM initiates rollback if configured at the ASDL level on the work order. The possible values are: • Y – Cancellation pending on the work order. • N – No cancellation received.
rollback_stat	number	1	0	The rollback status of the work order. If any CSDL on the work order requires rollback, this field is set to (201) RBACK_REQUIRED. Otherwise, it is set to (200) BACK_NOT_REQUIRED and no rollback takes place if the work order fails. If rollback is required, it starts on the work order and the SARM updates this field to (202) RBACK_IN_PROGRESS. This is a transient state and at the end of the rollback (Complete or Failed), the rollback status reverts back to its original state. If the rollback procedure fails, the field is updated to (204) RBACK_FAILED, otherwise (203) RBACK_COMPLETE to denote successful rollback. These values are defined in the header file sarm_defs.h.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
command	number	4	0	This field is transmitted by the SRP to the SARM and informs the SARM processing command to apply it to the work order. Possible values are: • ASAP_CMD_WO_UPDATE – The work order is either new or an update to an existing one. • ASAP_CMD_WO_CANCEL – The work order is a cancellation request on an existing order. • ASAP_CMD_WO_TRAN_ERROR – Indicates to the SARM that a translation error occurred on the work order. • ASAP_CMD_WO_HELD – The work order is to be held by the SARM until released by either the originating system with an update request or through the user interface. • ASAP_CMD_WO_REVIEW – The work order is to be held in a Reviewed state by the SARM until released by either the originating system through an update request or through the user interface. • ASAP_CMD_WO_REPLACE – Rollback an existing work order and submit with the same wo_id but a different set of data These values are defined in the header file asap_core.h.
crit_seq_no	number	4	1	The sequence number of the last critical update performed on this work order. This is updated each time a critical update to the work order has been performed.
lock_uid	varchar2	64	1	The user who last locked the work order for updating.
lock_dts	date	-	1	The date and time that the work order was last locked for updating.
start_dts	date	-	1	The date and time that the work order started provisioning.
asdl_timeout	number	4	1	If set, the ASDL time-out interval to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter ASDL_TIMEOUTS.
parent_wo	varchar2	80	1	If set, the parent work order on which the work order is dependent. The parent order must be completed for the work order to begin provisioning.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
wo_timeout	number	4	1	If set, the work order time-out interval to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter ORDER_TIMEOUT. The order timeout behavior is governed by two parameters: the wo_timeout parameter on the work order and the ORDER_TIMEOUT configuration parameter in ASAP.cfg. If wo_timeout has a value greater than one, it is used. If wo_timeout has a value of zero, work orders do not time out. If wo_timeout has a value less than zero, ORDER_TIMEOUT is used. If wo_timeout has a value less than zero and ORDER TIMEOUT has a value of zero or less than zero, work orders do not time out. The work order/ASDL timer starts after the work order has been submitted and the first ASDL starts provisioning. This threshold can be exceeded if, for example, the connection to an NE is interrupted after the connection has been established.
asdl_retry_num	number	4	1	If set, the number of ASDL retries to be used on the work order instead of the system-wide SARM default which is specified by the configuration parameter NUM_TIMES_RETRY.
asdl_retry_int	number	4	1	If set, the time period in seconds between ASDL retries to be used on the work order instead of the system-wide SARM default.
wo_rback	char	1	1	If set to: • (Y)es – A flag specifies whether to explicitly roll back the order in the event of failure. • (N)o – The order is not rolled back. • (D)efault – The SARM receives the setting from the SRP.
stub_update_req	-	-	-	-
asdl_delay_fail	char	1	1	If set, a flag which specifies whether or not to treat hard errors encountered on the work order as delayed failures, therefore allowing the work order to finish processing before failing. This is used to override the behavior of the NEP State Table.

Table 2-91 (Cont.) tbl_wrk_ord (SARM) Columns

Column_name	Type	Length	Nulls	Description
max_delay_fail	number	4	1	If set, the number of ASDL delayed failures to allow before failing a batch work order. This allows the overriding of the system-wide SARM default which is specified by the configuration parameter, BATCH_THRESHOLD.
srq_id	number	4	1	The ID of the latest SRQ associated with the work order. There is a single SRQ related to a given work order.
is_future_dated	char	1	1	If set to: • Y – It indicates that the work order is future-dated. • N – The work order is not future-dated. This is used by the ADMIN Server.
batch_group	varchar2	80	1	The batch group ID of the work order.
extsys_id	varchar2	128	1	An external system ID that is passed with the work order. It is not used by the SARM, but is used by the SRP for proper routing to upstream systems.
rollback_exceptions	char	1	0	The rollback exceptions flag returned by the SARM on the work order rollback completion notification. Indicates whether there are exceptions (i.e. ASDL failure) during rollback of a work order.
point_of_no_return	number	38	0	Values are: • -1 – No rollback if work order fails • 0 – (default). Normal rollback behavior. No 'point of no return' functionality. • >0 – ASDL_SEQ_NO. This ASDL is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this ASDL but no further.
failure_reason	varchar2	255	1	Provides the failure reason for a work order that fails during provisioning.

Table indexes:

```
Indexes

wo_id
wo_stat, sched_dts, org_unit
parent_wo
batch_group
```

temp_csdl_estim

A temporary storage table for the Oracle **SSP_csdl_list** function. Data from this table is automatically maintained by the function; therefore, you should not manually add or remove data.

Table 2-92 temp_csdl_estim Columns

Column_name	Type	Length	Nulls	Description
sess_id	number	22	1	Oracle session ID of the session executing the function.
csdl_seq_no	number	4	1	CSDL sequence number.
estimate	number	4	1	CSDL estimate.

Table indexes:

```
sess_id
```

temp_csdl_list

A temporary storage table for the Oracle **SSP_csdl_list** function. Data from this table is automatically maintained by the function; therefore, you should not manually add or remove data.

Table 2-93 temp_csdl_list Columns

Column_name	Type	Length	Nulls	Description
sess_id	number	22	1	Oracle session ID of the session executing the function.
csdl_cmd	varchar2	80	1	CSDL command.
csdl_st	number	1	1	CSDL status.
csdl_seq_no	number	4	1	CSDL sequence number.
csdl_id	number	4	1	CSDL ID.
description	varchar2	255	1	Description of the CSDL.
prov_sequence	number	4	1	Provisioning sequence of the CSDL.
asdl_seq_no	number	4	1	ASDL sequence number.

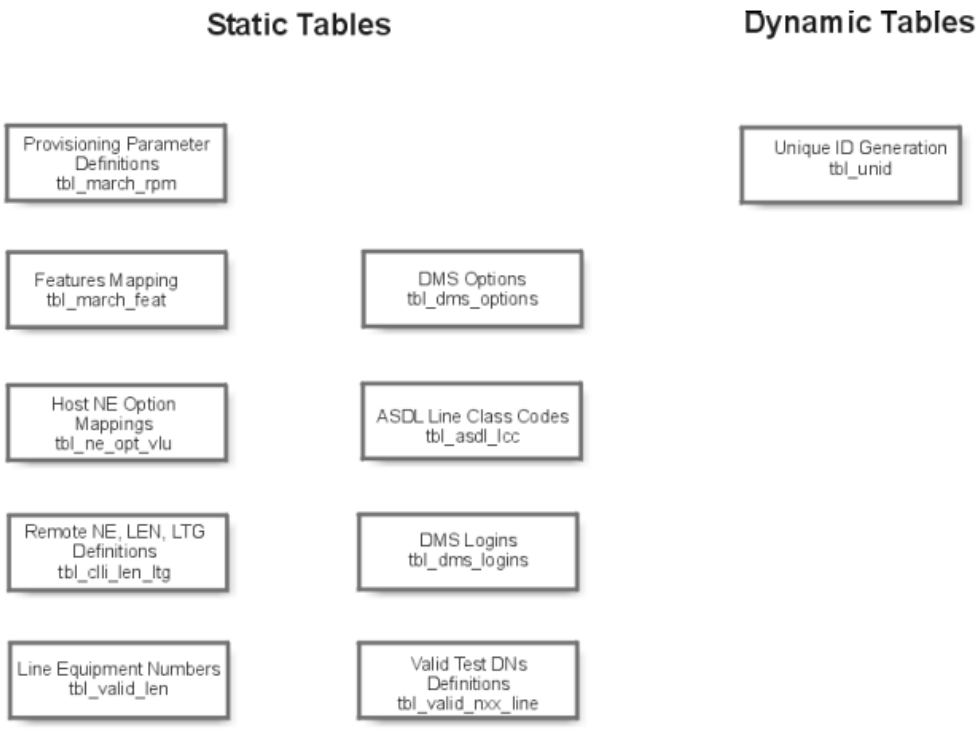
Table indexes:

```
sess_id
```

NEP database

This section details the NEP Database tables.

Figure 2-1 NEP database tables



User-created database tables

Following is a list of user-created database tables.

tbl_asdl_lcc

This static table contains the line class codes for certain ASDLs. It is only used by NE State Table queries.

Table 2-94 tbl_asdl_lcc Columns

Column_name	Type	Length	Nulls	Description
asdl	char	80	0	The ASDLs that add an access line.
lcc	char	5	0	The Line Class Code (LCC) for a specific ASDL adding an access line.

Table indexes:

`asdl`

tbl_clli_len_ltg

This static table contains specific line information. For example, site code, line treatment group, intercept information, etc. This table is referenced by the GET_INCPT and GET_LTG State Table actions.

Table 2-95 tbl_clli_len_ltg Columns

Column_name	Type	Length	Nulls	Nulls
mach_clli	varchar2	128	0	Remote NE.
nxx	varchar2	3	0	The NXX or Exchange working from this Remote NE.
site	char	4	1	Site code associated with this Remote NE and NXX.
ltg	number	38	1	Line treatment group for determining if calls are toll calls.
pub_incpt	char	4	1	Indicates if the intercept on the line is for a directory-published number.
non_pub_incpt	char	4	1	Indicates if the intercept on the line is for a non-published directory number.
clli_desc	varchar2	255	1	Description on the Remote NE.
from_len	varchar2	7	0	Start of LEN (Line Equipment Number) range.
to_len	varchar2	7	0	End of LEN range.
host_clli	char	80	0	The Host NE managing the Remote NE.

Table indexes:

```
Indexes

mach_clli, nxx, from_len
host_clli, site, from_len
```

tbl_dms_logins

This static table is used by the NEP and contains the login user IDs and passwords for the connections to the DMS NEs. You are responsible for populating and maintaining this table.

Table 2-96 tbl_dms_logins Columns

Column_name	Type	Length	Nulls	Description
host_clli	char	80	0	Host NE.
login_id	char	30	0	The login user ID that ASAP uses for this Host NE.
password1	char	30	0	The first NE password in a sequence of two passwords which are alternated.
password2	char	30	0	The second NE password.
last_changed	date	-	0	Date and time that the password was last changed.

Table indexes:

```
host_clli, login_id
```

tbl_dms_options

This static table is used to specify DMS specific options.

Table 2-97 tbl_dms_options Columns

Column_name	Type	Length	Nulls	Description
tech	char	16	0	The technology of the Host NE.
sftwr_load	char	16	0	The software load of the Host NE.
dms_option	varchar2	10	0	The DMS option.

Table indexes:

```
tech, sftwr_load, dms_option
```

tbl_march_feat

This static table provides the ability to map a generic option to a switch specific option. It is referenced by the GET_SW_FEAT State Table action.

Table 2-98 tbl_march_feat Columns

Column_name	Type	Length	Nulls	Description
generic_feat	varchar2	20	0	The generic feature.
tech	varchar2	16	0	The technology of the Host NE.
switch_feat	varchar2	20	0	The switch specific option.

Table indexes:

```
generic_feat, tech
```

tbl_march_rpm

This static table provides the NEP State Tables access to provisioning parameters that are specific to a particular Host NE. The action, GET_P_PARMS, uses a variety of action strings that depend on the provisioning parameters being accessed.

Only the values of the parameters (ASDL parameters or State Table program local variables) are used, therefore the order of the parameters is fixed. The following is the order of the various parameter types:

- (%TYPE, %MCLI)
- (%TYPE, %MCLI, %NXX)
- (%TYPE, %MCLI, %USOC)
- (%TYPE, %MCLI, %USOC, %NXX)
- (%TYPE, %MCLI, %FEAT)
- (%TYPE, %MCLI, %PIC)
- (%TYPE, %MCLI, %USOC, %LCC)

- (%TYPE, %MCLI, %USOC, %NXX, %LCC)

Table 2-99 tbl_ne_opt_vlu Columns

Column_name	Type	Length	Nulls	Description
host_clli	char	80	0	The Host NE.
param_type	varchar2	10	0	The type of provisioning parameter. The following values include: • H – Host parameters. • HN – Host/nxx parameters. • HU – Host/usoc parameters. • HUN – Host/usoc/nxx parameters. • HF – Host/feature parameters. • HP – Host pic conversion. • HUL – Host/usoc/lcc parameters. • HUNL – Host/usoc/nxx/lcc parameters. • RCCF – Remote activation CCF's for a 5ESS host. • NACT – No activate CCF's for a 5ESS host.
usoc	varchar2	10	1	The recent change Universal Service Order Code (USOC).
nxx	char	3	1	The NXX (Network Number Exchange).
lcc	varchar2	10	1	The line class code.
feat	varchar2	20	1	Switch feature name.
param_lbl	varchar2	80	0	Parameter label.
param_vlu	varchar2	30	1	Parameter value.

Table indexes:

```
host_clli, param_type, usoc, nxx, lcc, feat, param_lbl
```

tbl_ne_opt_vlu

This static table is used by the State Tables. It contains the option string to be sent to the Network Element.

Table 2-100 ttbl_ne_opt_vlu Columns

Column_name	Type	Length	Nulls	Description
tech	char	255	0	Technology of the Host Network Element.
sftwr_load	char	16	0	Software Version of the Host Network Element
asdl_cmd	char	80	0	The ASDL Command for which the option string needs to be sent to the NE.
mask_lbl	char	10	1	A masking label for the option format
opt_vlu	char	80	1	Option Value
ne_opt_vlu	char	80	0	The option value for the particular NE Type and Software Load.

Table indexes:

```
tech, sftwr_load, asdl_cmd, mask_lbl, opt_vlu
```

tbl_unid

This dynamic table provides a method of generating a serial field. You can manage unique IDs required by other tables. It is present in most user-created databases.

Table 2-101 tbl_unid Columns

Column_name	Type	Length	Nulls	Description
unid_type	varchar2	12	0	A unique code identifying the UNID type. This allows many different UNID values for different types of UNID.
unid	number	38	0	The UNID value for the type.
pad1	char	255	0	Padding to make a table row occupy a page. This is to reduce concurrence on the database data page by different database processes.
pad2	char	255	0	Same as pad1.
pad3	char	255	0	Same as pad1.
pad4	char	255	0	Same as pad1.

Table indexes:

```
unid_type
```

tbl_valid_len

This static table contains a set of line equipment numbers that ASAP accesses in a non-production environment. This table is checked from the CHECK_DATA Interpreter ASDL when the ASAP_SYS environment variable does not equal PROD, and the Interpreter is not in loopback mode.

The CHECK_DATA State Table program and tbl_valid_len database table are included as samples so that you may create your own data checking tables.

tbl_valid_len is used to safeguard line equipment numbers (other than those listed in this table) when system testing is being conducted on a production NE. If validation is requested and a particular LEN is not found in this check table, then the order is failed.

Table 2-102 tbl_valid_len Columns

Column_name	Type	Length	Nulls	Description
mcli	char	80	0	Remote NE of the LEN.
len	char	7	0	Valid testing Line Equipment Number (LEN).

Table indexes:

```
mcli, len
```

tbl_valid_nxx_line

This static table contains a set of telephone numbers that the ASAP system accesses in a non-production environment. This is a search table for the CHECK_DATA Interpreter State Table program when the ASAP_SYS environment variable does not equal PROD and the Interpreter is not in loopback mode.

The CHECK_DATA State Table program and tbl_valid_nxx_line database table are included as samples so that you may create your own data checking tables.

This table is to safeguard telephone numbers (other than those listed in this table) when system testing is being conducted on a production NE. If validation is requested and a particular DN is not found in this check table, the ASDL is failed.

Table 2-103 tbl_valid_nxx_line Columns

Column_name	Type	Length	Nulls	Description
nxx	char	3	0	Valid test NXX.
line	char	4	0	Valid test line.

Table indexes:

```
nxx, line
```

Admin database

This section describes the Admin database tables.

In addition to the user-created database tables described in the following section, the Admin database contains a **WLStore** table that is populated and managed by WebLogic Server to maintain persistence in their JMS destinations. It is recommended that you manually clean this tables during development if you want to start with a fresh environment. Otherwise, old undelivered messages occupy the queues and may cause problems when trying to debug or test. For production environments, you should design applications to handle undelivered messages. For instance, you can configure an error destination to use if a message fails to be delivered after a configurable number of attempts.

For more information on either of these two tables, refer to WebLogic Server documentation.

User-created database tables

Following is a list of user-created database tables.

tbl_asap_sarm

Defines the relationship between which Admin servers monitor which type of SARM servers.

Table 2-104 tbl_asap_sarm Columns

Column_name	Type	Length	Nulls	Description
adm_svr	char	8	0	Name of the Admin server.

Table 2-104 (Cont.) tbl_asap_sarm Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server.

Table indexes:

```
adm_svr, sarm
```

tbl_oca_svr

This table provides details about the number of active sessions maintained by the old CORBA SRP Server, if the load balancing feature is enabled. The new OCA SRP does not use this table.

Table 2-105 tbl_oca_svr Columns

Column_name	Type	Length	Nulls	Description
svr_name	Varchar2	8	0	Name of the CORBA SRP Server.
host_name	Varchar2	80	0	The name of the host on which the CORBA SRP server application resides.
session_no	Number	38	0	The number of sessions that the CORBA SRP Server is holding.
status	Varchar2	16	0	Status of the sessions.

tbl_perf_asdl

Contains ASDL performance information retrieved from the SARM server in the ASAP system which continually maintains the data in memory. The frequency of the retrieval is controlled by the POLL_TIMER_ASDL configuration parameter.

You can view CSDL performance information using the asap_utils function: 22. Admin - ASDL Stats. For more information, refer to the *ASAP Server Configuration Guide.*

Table 2-106 tbl_perf_asdl Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server providing the data.
date_recvd	date	-	0	The date and time stamp of the record.
time_elapse	float	8	0	Number of seconds since last performance data was received.
record_type	char	1	0	Type of record recorded. Possible values include: • P – Poll record • S – Summary record
asdl_cmd	char	80	1	The ASDL command.
num_execute	number	4	0	Number of ASDL executions.

Table 2-106 (Cont.) tbl_perf_asdl Columns

Column_name	Type	Length	Nulls	Description
num_failed	number	4	0	Number of ASDL failures.
num_complete	number	4	0	Number of successful ASDL completions.
parm_avg	float	8	0	Average number of parameters on the ASDL.
parm_min	number	4	0	Minimum number of parameters on the ASDL.
parm_max	number	4	0	Maximum number of parameters on the ASDL.
num_rollback	number	4	0	Number of times the ASDL was rolled back.
num_soft_err	number	4	0	Number of soft errors on the ASDL.
num_retries	number	4	0	Number of times the ASDL was executed.
num_skipped	number	4	0	Number of times the ASDL was skipped in processing.
comp_tm_avg	number	8	0	The average time in seconds for the ASDL to complete.
comp_tm_min	float	8	0	The minimum time in seconds for the ASDL to complete.
comp_tm_max	float	8	0	The maximum time in seconds for the ASDL to complete.
parm_count	number	4	1	Number of ASDL parameters.
comp_tm	float	8	1	Total ASDL processing time in seconds.

Table indexes:

```
sarm, asdl_cmd, date_recvd, record_type
```

tbl_perf_csdl

Contains CSDL performance information retrieved from the SARM server in the ASAP system which continually maintains the data in memory. The frequency of the retrieval is controlled by the POLL_TIMER_CSDL configuration parameter.

You can view CSDL performance information using the asap_utils function: 21. Admin - CSDL Stats. For more information on asap_utils, refer to the *ASAP Server Configuration Guide.*

Table 2-107 tbl_perf_csdl Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server providing the data.
date_recvd	date	-	0	The date and time stamp of the record.
time_elapse	float	8	0	Number of seconds since last performance data was received.

Table 2-107 (Cont.) tbl_perf_csdl Columns

Column_name	Type	Length	Nulls	Description
record_type	char	1	0	Type of record recorded. Possible values include: • P – Poll record • S – Summary record
csdl_cmd	char	80	1	The CSDL command.
num_received	number	4	0	Number of times the CSDL was received by the SARM.
num_complete	number	4	0	Number of successful CSDL completions.
num_failed	number	4	0	Number of CSDL failures.
comp_tm_avg	float	8	0	Average completion time in seconds of the CSDL.
comp_tm_min	float	8	0	Minimum completion time in seconds of the CSDL.
comp_tm_max	float	8	0	Maximum completion time in seconds of the CSDL.
num_asdl_comp	number	4	0	Number of completed ASDLs on the CSDL.
num_asdl_skip	number	4	0	Number of skipped ASDLs on the CSDL.
parms_avg	float	8	0	Average number of CSDL parameters.
parms_min	number	4	0	Minimum number of CSDL parameters.
parms_max	number	4	0	Maximum number of CSDL parameters.
asdl_comp_avg	float	8	0	Average number of completed ASDLs on the CSDL.
asdl_comp_min	number	4	0	Minimum number of completed ASDLs on the CSDL.
asdl_comp_max	number	4	0	Maximum number of completed ASDLs on the CSDL.
asdl_skip_avg	float	8	0	Average number of ASDLs skipped on the CSDL.
asdl_skip_min	number	4	0	Minimum number of ASDLs skipped on the CSDL.
asdl_skip_max	number	4	0	Total number of ASDLs skipped on the CSDL.
asdl_comp_tot	number	4	1	Total ASDL Processing time.
asdl_comp_count	number	4	1	Number of ASDLs completed.
asdl_skip_count	number	4	1	Number of ASDLs skipped.
comp_tm_tot	number	4	1	Total ASDL processing time.
parm_count_tot	number	4	1	Total number of CSDL parameters.

Table indexes:

```
sarm, csdl_cmd, date_recvd, record_type
```

tbl_perf_ne

Contains NE performance information retrieved from the SARM server in the ASAP system which continually maintains the data in memory. The frequency of the retrieval is controlled by the POLL_TIMER_NE configuration parameter.

You can view NE performance information using the asap_utils function: 23. Admin - NE Stats. For more information on asap_utils, refer to the *ASAP Server Configuration Guide.*

Table 2-108 tbl_perf_ne Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server providing the data.
date_recvd	date	-	0	The date and time stamp of the record.
time_elapse	float	8	0	Number of seconds since the last performance data was received.
record_type	char	1	0	Type of record. Possible values include: • P – Poll record • S – Summary record
nep	char	8	1	The NEP server.
host_clli	varchar2	80	1	The Host NE.
tech	char	16	1	The technology of the Host NE.
sftwr_load	char	16	1	The software load of the Host NE.
state	char	10	1	The Host NE state.
num_estimate	number	4	0	The ASDL estimate for the NE.
num_pending	number	4	0	Number of ASDLs in the Pending queue to the Host NE.
num_in_prog	number	4	0	Number of ASDLs in the In Progress queue to the Host NE.
num_connect	number	4	0	Number of connections to the Host NE.
num_retry	number	4	0	Number of ASDLs in the Retry queue to the Host NE.
time_tot_avail	float	8	0	Total time in seconds the NE is available.
time_curr_avail	float	8	0	Current time in seconds the NE has been available.
num_asdl_comp	number	4	0	Number of ASDLs completed to the Host NE.
num_asdl_fail	number	4	0	Number of ASDLs failed to the Host NE.
num_asdl_retry	number	4	0	Number of ASDLs retried to the Host NE.
num_asdl_recvd	number	4	0	Number of ASDLs received for the Host NE.
num_asdl_xfer	number	4	0	Number of ASDLs transferred.
perc_ne_usage	float	8	0	Percentage of time that ASAP is provisioning the Host NE while it has open connections to the Host NE.
asdl_comp_tm_avg	float	8	0	Average time in seconds required to complete an ASDL to the Host NE.

Table 2-108 (Cont.) tbl_perf_ne Columns

Column_name	Type	Length	Nulls	Description
asdl_comp_tm_min	float	8	0	Minimum time in seconds required to complete an ASDL to the Host NE.
asdl_comp_tm_max	float	8	0	Maximum time in seconds required to complete an ASDL to the Host NE.
asdl_q_avg	float	8	0	Average ASDL Pending queue size to the Host NE.
asdl_q_min	number	4	0	Minimum ASDL Pending queue size to the Host NE.
asdl_q_max	number	4	0	Maximum ASDL Pending queue size to the Host NE.
pend_q_avg	float	8	0	Average number of ASDLs in the SARM.
pend_q_max	number	4	0	Maximum number of ASDLs in the SARM.
time_tot_maint	float	8	0	Total time in seconds the Host NE is in Maintenance mode.
time_curr_maint	float	8	0	Current time in seconds that the NE is in Maintenance mode.
comp_tm_tot	float	8	1	Total ASDL processing time.
q_time_tot	float	8	1	Total ASDL queue time.
tot_pq_sz	number	4	1	Total Pending queue size.
pq_samples	number	4	1	Number of Pending queue samples.

Table indexes:

```
sarm, host_clli, date_recvd, record_type
```

tbl_perf_ne_asdl

Collects the frequency of exit conditions for the various NEs and ASDLs. The frequency of the retrieval is controlled by the POLL_TIMER_NE_ASDL configuration parameter.

You can view pseudo real-time statistical information related to SARM NE / ASDL processing using the asap_utils function: 24. Admin - NE/ASDL Stats. For more information on asap_utils, refer to the *ASAP Server Configuration Guide.*

Table 2-109 tbl_perf_ne_asdl Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server providing the data.
date_recvd	date	-	0	The date and time stamp of the record.
time_elapse	float	8	0	Number of seconds since the last performance data was received.

Table 2-109 (Cont.) tbl_perf_ne_asdl Columns

Column_name	Type	Length	Nulls	Description
record_type	char	1	0	Type of record. Possible values include: • P – Poll record • S – Summary record
host_clli	varchar2	80	1	The host NE where the ASDL command is routed by the SARM. Although one host NE is specified on the CSDL, different ASDLs on the CSDL is routed to different Host NEs based on the ASDL command.
asdl_cmd	char	80	1	The ASDL command.
user_exit_type	char	20	1	The user-defined ASDL exit type.
counter	number	4	0	The number of times the user_exit_type occurred.

Table indexes:

```
sarm, host_clli, asdl_cmd, user_exit_type, date_recvd, record_type
```

tbl_perf_order

Contains work order performance information retrieved from the SARM server in the ASAP system which continually maintains the data in memory. The frequency of the retrieval is controlled by the POLL_TIMER_ORDER configuration parameter.

You can view work order performance information using the asap_utils function: 20. Admin - WO Stats. For more information on asap_utils, refer to the *ASAP Server Configuration Guide.*

Table 2-110 tbl_perf_order Columns

Column_name	Type	Length	Nulls	Description
sarm	char	8	0	Name of the SARM server providing data.
date_recvd	date	-	0	The date and time stamp of the record.
time_elapse	float	8	0	Number of seconds since the last performance data was received.
record_type	char	1	0	Type of record recorded. Possible values include: • P – Poll record • S – Summary record
org_unit	char	8	1	Organization unit associated with the work order.
ord_type	char	1	1	Work order type.
num_recorded	number	4	0	Number of work orders recorded in the SARM.
num_cancelled	number	4	0	Number of cancelled work orders.
num_received	number	4	0	Number of received work orders.
num_future	number	4	0	Number of future-dated work orders.

Table 2-110 (Cont.) tbl_perf_order Columns

Column_name	Type	Length	Nulls	Description
num_immed_fail	number	4	0	Number of immediate failed work orders.
num_immed_comp	number	4	0	Number of immediate completed work orders.
num_tran_err	number	4	0	Number of translation errors received.
num_timeout	number	4	0	Number of timed out work orders.
num_rollback	number	4	0	Number of rolled back work orders.
num_update	number	4	0	Number of work orders updated after the original was sent.
num_collision	number	4	0	Number of work order collisions.
num_route_err	number	4	0	Number of work orders that have routing errors.
global_p_avg	float	8	0	Average number of global parameters on the work order.
global_p_min	number	4	0	Minimum number of global parameters on the work order.
global_p_max	number	4	0	Maximum number of global parameters on the work order.
latency_avg	float	8	0	Average work order replication latency in seconds.
latency_min	number	4	0	Minimum work order replication latency in seconds.
latency_max	number	4	0	Maximum work order replication latency in seconds.
num_fut_fail	number	4	0	The number of future-dated work orders that have failed.
num_fut_comp	number	4	0	The number of future-dated work orders that have completed.
latency_tot	number	4	1	Total work order replication latency in seconds.
latency_count	number	4	1	Number of work order replication latencies that have occurred.
global_p_tot	number	4	1	Total number of global parameters on all work orders.
global_p_count	number	4	1	Total number of global parameters that have occurred.

Table indexes:

```
sarm, org_unit, ord_type, date_recvd, record_type
```

tbl_aims_msg_convert

This table stores translations of all localizable ASAP strings to the destination languages.

Table 2-111 tbl_aims_msg_convert Columns

Column_name	Type	Length	Nulls	Description
lang_cd	varchar2	32	0	The language code.
msg_id	varchar2	64	0	Unique message identifier to identify the messages referenced in other tables.
type	varchar2	16	0	Specifies the type of message, such as CSDL description, ASDL description, etc.
message	varchar2	255	0	The message test.
var_description	varchar2	40	1	Description of the message, for example, the substitutable fields within it.

tbl_aims_preference

This dynamic table stores preferences in the OCA client.

Table 2-112 tbl_aims_preference Columns

Column_name	Type	Length	Nulls	Description
user_id	varchar2	30	0	User ID.
name	varchar2	100	0	Name of the preference.
value	varchar2	200	1	Value for the preference. You are responsible to convert the preference to a string.

Table indexes:

```
user_id, name
```

tbl_aims_map_acl (Not used)

Not used.

tbl_aims_audit_log (Not used)

Not used.

tbl_aims_component

Not used.

tbl_aims_function

Not used.

tbl_aims_operation

Not used.

tbl_aims_param

Not used.

tbl_aims_rpc

Not used.

tbl_aims_rpc_defn

Not used.

tbl_aims_rpc_dest

Not used.

tbl_aims_rpc_dest_defn

Not used

tbl_aims_rpc_param

Not used.

tbl_aims_rpc_param_defn

Not used.

tbl_aims_rpc_param_type

Not used.

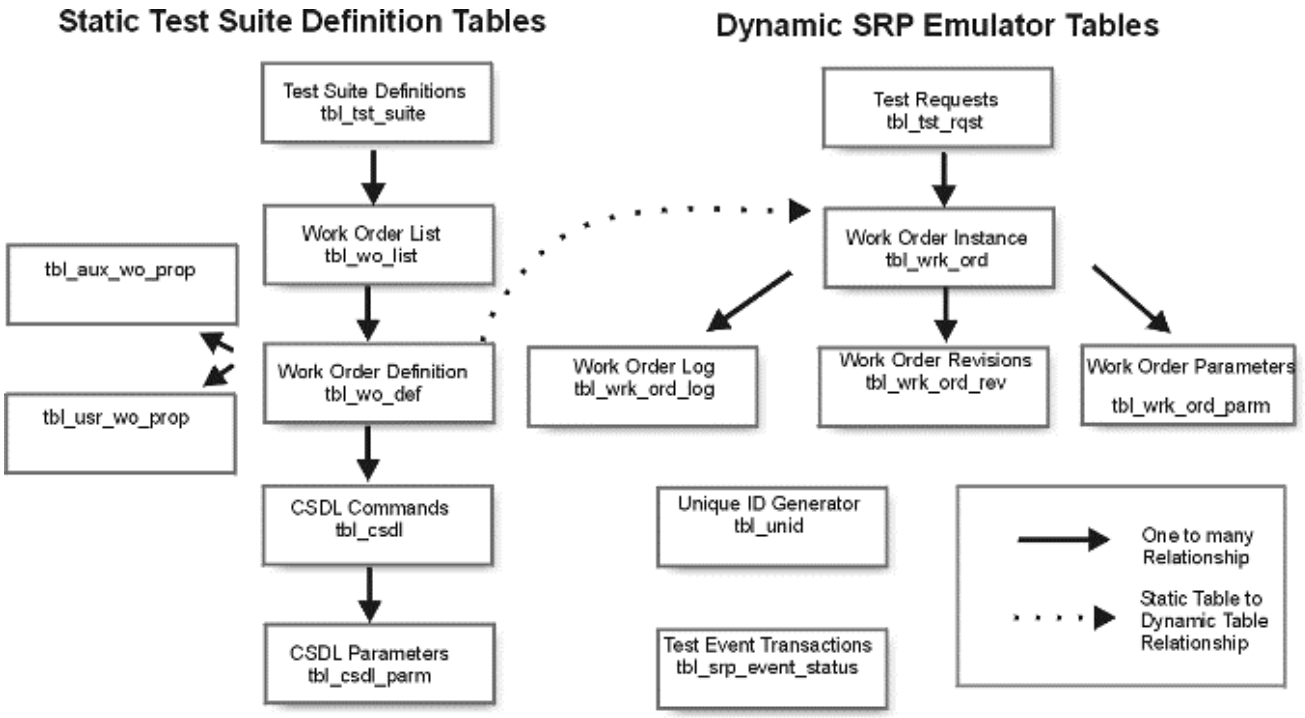
tbl_aims_template

Not used.

C++ SRP API emulator database

SRP emulator is used for performance benchmarking, system testing, and the prototyping of new SRPs. The SRP emulator is a generic SRP that submits work orders to the SARM. The SRP emulator makes use of the SRP, Interpreter, Server Application, and Common API libraries.

Figure 2-2 C++ SRP API Emulator Database Tables



> **Note:**
>
> If you shut down and restart the C++ SRP API Emulator when USE_RAW_WO_ID is set to 0, it may result in a unique work order name containing a recurring prefix.

User-created database tables

Following is a list of user-created database tables.

tbl_aux_wo_prop

tbl_aux_wo_prop is a class A dynamic table that serves as an extension to tbl_wrk_ord. tbl_aux_wo_prop was designed to accommodate additional pre-defined work order properties to supplement the ones contained in tbl_wrk_ord. Currently, the only extended property supported in tbl_aux_wo_prop is WO_SECURITY_PROP. This property is maintained for each work order. If WO_SECURITY_PROP = 0, then work order information is eligible to be output to diagnostic files. If WO_SECURITY_PROP = 1, then no work order information is written to diagnostic files.

Refer to the *ASAP System Administrator's Guide* for more information on secure work order information.

When the work orders in the tbl_wrk_ord are deleted, the corresponding records in tbl_aux_wo_prop are deleted. This table can be purged using function **SSP_db_admin**. For guidelines and instructions on database purging, refer to the *ASAP System Administrator's Guide.*

Table 2-113 tbl_aux_wo_prop

Column Name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The ID of the work order property.
name	varchar2	80	0	The name of the work order property.
value	varchar2	255	1	The value given to the work order property.

Table indexes:

```
Unique

wo_id, name
```

tbl_csdl

Used only in conjunction with the SRP Emulator.

This static table contains all CSDLs for each logical work order listed in the tbl_wo_def table. Each logical work order can have one or more associated CSDLs.

Table 2-114 tbl_csdl

Column_name	Type	Length	Nulls	Description
wo_def_id	char	80	0	Unique ID of the logical work order associated with the CSDL.
csdl_seq_no	number	38	0	Sequence number of the CSDL within the logical work order.
csdl_cmd	char	80	0	The CSDL command.
csdl_st	number	38	0	Initial status of the CSDL that gets transmitted to the SARM along with the rest of the work order details. Possible values include: • (50) ASAP_CSDL_INITIAL – CSDL ready for provisioning. • (51) ASAP_CSDL_HELD – CSDL is to be held by the SARM, not provisioned. • (52) ASAP_CSDL_MANUAL_TRAN – CSDL is later manually translated by a user. • (53) ASAP_CSDL_TRAN_ERR – A translation error has occurred on this CSDL. Values are defined in the header file asap_core.h.

Table indexes:

```
Unique

wo_def_id, csdl_seq_no
```

tbl_csdl_parm

This static table contains all CSDL parameters associated with a specific CSDL.

Table 2-115 tbl_csdl_parm

Column_name	Type	Length	Nulls	Description
wo_def_id	char	80	0	The logical work order ID.
csdl_seq_no	number	38	0	Sequence number of the CSDL within the logical work order.
parm_seq_no	number	38	0	Sequence number of the CSDL parameter within the CSDL.
parm_lbl	varchar2	80	0	CSDL parameter label; for example, DN, EN, etc.
parm_vlu	varchar2	255	0	CSDL parameter value associated with the parameter label.

Table indexes:

```
Unique

wo_def_id, csdl_seq_no, parm_seq_no
```

tbl_srp_event_status

This dynamic table contains work order status information.

Table 2-116 tbl_srp_event_status

Column_name	Type	Length	Nulls	Description
wo_def_id	char	80	0	Logical work order ID.
event_name	varchar2	50	0	Name of the event to test.
event_status	number	38	0	The status that is assigned to the event. • 1 - Succeed • 0 - Fail

Table indexes:

```
Unique

wo_def_id, event_name
```

tbl_tst_rqst

This dynamic table contains all test requests sent to the C++ SRP API Emulator. It contains test details to be used by the C++ SRP API Emulator, such as the test suite to use, type of test to run, number of instances of each work order in the test suite, etc. A test request is an instance of a test suite.

Table 2-117 tbl_tst_rqst

Column_name	Type	Length	Nulls	Description
tst_rqst_id	number	38	0	Unique test request ID.
tst_suite_cd	char	96	0	Test suite code that specifies which test suite to use.
sync_test	char	1	0	Specifies whether the test is synchronous, for example, waiting for test completion, or asynchronous. Possible values are: • Y – Synchronous test. • N – Asynchronous test.
repeat_cnt	number	38	0	Specifies how many instances are created of each logical work order in the test suite and runs stress tests of the system.
all_wos_sent	char	1	0	A flag indicating whether all work orders associated with the test request have been sent to the SARM. You can use this field to determine if the test is complete, for example, if all_wos_sent = “Y" and wo_cnt = wo_cmp_cnt, the test is complete. Possible values are: • N – More orders are still to be sent to the SARM. • Y – All orders were sent to the SARM.
wo_cnt	number	38	1	Number of work orders in the test request. To be initialized to 0 by the user. The C++ SRP API Emulator updates the count as each work order is created and the wo_cnt determines if the test is complete.
wo_cmp_cnt	number	38	1	A count of the completed work orders in the test. You must set this to 0. The C++ SRP API Emulator updates the count as each work order is completed and determines if the test is complete.
tst_userid	char	32	1	The ID of the user. This field is only for diagnostic purposes.

Table indexes:

```
Unique

tst_rqst_id
```

tbl_tst_suite

You can use this static table to create test suite definitions. Each test suite has a list of logical work orders associated with it. A test request is an instance of a test suite.

Table 2-118 tbl_tst_suite

Column_name	Type	Length	Nulls	Description
tst_suite_cd	char	96	0	Identifies a test suite. This column is a foreign key in tst_rqst.
tst_suite_desc	varchar2	255	1	A description of a test suite.
create_dt	date	-	0	Test suite creation date.

Table indexes:

```
Unique

tst_suite_cd
```

tbl_unid

Use this dynamic table to manage unique IDs required by other tables in this database. It is present in most created databases and provides a method of generating a serial field.

Table 2-119 tbl_unid

Column_name	Type	Length	Nulls	Description
unid_type	char	32	0	A unique code identifying the UNID type. This allows many different UNID values for different types of UNID.
unid	number	38	0	The UNID value for that particular type.
pad1	char	255	0	Padding to make a table row occupy a database page. This reduces concurrence on the database data page by different database processes.
pad2	char	255	0	Same as pad1.
pad3	char	255	0	Same as pad1.
pad4	char	255	0	Same as pad1.

Table indexes:

```
Unique

unid_type
```

tbl_usr_wo_prop

This table is used to dynamically configure a work order property. It consists of WO ID and name/value pair for a user-defined work order property, and is used mainly for work order queries. This table supports only string type user-defined work order properties (also known as extended work order properties). Extended work order properties are only applicable to the work order where they are defined.

Table 2-120 tbl_usr_wo_prop

Column Name	Type	Length	Nulls	Description
wo_id	varchar2	80	0	The ID of the work order property.
name	varchar2	80	0	The name of the work order property.
value	varchar2	255	1	The value given to the work order property.

Table indexes:

```
Unique

wo_id, name
```

tbl_wo_def

This static table contains all logical work orders in the C++ SRP API Emulator database.

Table 2-121 tbl_wo_def

Column_name	Type	Length	Nulls	Description
wo_def_id	char	80	0	Logical work order ID.
wo_def_desc	varchar2	255	1	A description of the work order definition.
origin	varchar2	64	0	The user who created the logical work order.
org_unit	char	8	0	Organization Unit for the work order. This is set to the C++ SRP API Emulator Application server name.
srp_stat	char	1	0	The SRP state of the logical work order. You can use this field to determine the operation field to transmit from the SRP to the SARM on the wo_begin RPC. You can find the field definitions in the local header file, srp_emul.h.
priority	char	1	0	The provisioning priority. Possible values include: • (1) ASAP_SRQ_HIGH_PRIO – High priority. • (5) ASAP_SRQ_NORMAL_PRIO – Normal priority. • (9) ASAP_SRQ_LOW_PRIO – Low priority. Values are defined in the header file asap_core.h.

Table 2-121 (Cont.) tbl_wo_def

Column_name	Type	Length	Nulls	Description
srq_type	char	1	0	Service Request type. Possible values include: • (A) ASAP_ADD_SRQ – Addition of service. • (R) ASAP_REMOVE_SRQ – Removal of service. • (C) ASAP_CHANGE_SRQ – Change/ update of service. • (Q) ASAP_QUERY_SRQ – Query only; no provisioning to be performed. Values are defined in the header file asap_core.h.
userid	varchar2	64	0	Specifies the user ID to be used for SARM security authorization. This is required only if the SARM security checking logic is currently configured.
password	char	30	0	Specifies the password to be used for SARM security authorization.
asdl_timeout	number	38	0	Specifies the ASDL time-out interval to be used on the work order provided that the configuration parameter ASDL_TIMEOUTS is set to 1. If the value of the ASDL_TIMEOUTS is zero (0), the ASDL timeout feature is not used, regardless of the asdl_timeout value. This value is passed to the SARM in the work order transaction to allow the specification of a work order dependent ASDL time-out interval.
parent_wo	char	80	1	If set, the parent work order upon which this work order is dependent. This work order will not begin provisioning until the parent work order has been completed.

Table 2-121 (Cont.) tbl_wo_def

Column_name	Type	Length	Nulls	Description
wo_timeout	number	38	1	If set, the work order time-out interval in seconds to be used on the work order instead of the system-wide SARM default (specified by the configuration parameter ORDER_TIMEOUT in ASAP.cfg). The order timeout behavior is governed by two parameters: the wo_timeout parameter on the work order and the ORDER_TIMEOUT configuration parameter in ASAP.cfg. If wo_timeout has a value greater than one, it is used. If wo_timeout has a value of zero or less than zero, ORDER_TIMEOUT is used. If wo_timeout has a value of zero or less than zero and ORDER TIMEOUT has a value of zero or less than zero, work orders do not time out. The work order/ASDL timer starts after the work order has been submitted and the first ASDL starts provisioning. This threshold can be exceeded if, for example, the connection to an NE is interrupted after the connection has been established.
asdl_retry_num	number	38	1	If set, the number of ASDL retries used on the work order instead of the system-wide SARM default.
asdl_retry_int	number	38	1	If set, the time in seconds between retries of an ASDL used on this work order instead of the system-wide SARM default.
wo_rback	char	1	1	Possible values: • Y – Work order rolls back if it times out or fails. • N – Work order does not roll back. If a value is not specified, the default is D. This means that the rollback depends on the CSDL parameter rollback_req. If rollback_req is set for a CSDL, then the work order rolls back when it times out or fails
propagate	char	1	1	Reserved.
asdl_delay_fail	char	1	1	If set, a flag specifying whether to treat hard errors encountered on the work order as delayed failures. The order then finishes processing before failing. You can use this field to override the behavior of the NEP State Table.
delay_threshold	number	38	1	If set, the number of ASDL delayed failures given before failing a batch work order. This permits the overriding of the system-wide SARM default.

Table 2-121 (Cont.) tbl_wo_def

Column_name	Type	Length	Nulls	Description
batch_group	char	80	1	Batch group for the work order.
extsys_id	varchar2	128	1	ID of the source of the work order (optional).

Table indexes:

```
wo_def_id
```

tbl_wo_list

This static table contains all logical work orders associated with each test suite listed in tbl_tst_suite.

Table 2-122 tbl_wo_list

Column_name	Type	Length	Nulls	Description
tst_suite_cd	char	96	0	The test suite code to which the logical work order list belongs.
wo_list_seq_no	number	38	0	Sequence number of the logical work order within the test suite.
wo_def_id	char	80	0	Logical work order ID.
delay	number	38	0	The delay in seconds after the transmission of the work order to the SARM. This allows the specification of time intervals between different work orders or copies of the same order.
operation	number	38	0	The work order operation to be transmitted to the SARM. Possible values include: • (1) ASAP_CMD_WO_UPDATE – The work order is either a new work order or an update to an existing one. • (2) ASAP_CMD_WO_CANCEL – The work order is a cancellation request on an existing work order. • (3) ASAP_CMD_WO_TRAN_ERROR – Indicates to the SARM that a translation error occurred. • (4) ASAP_CMD_WO_HELD – The work order is to be held by the SARM until released by either the originating system with an update request or through the user interface. • (5) ASAP_CMD_WO_REVIEW – The work order is to be held in a Reviewed state by the SARM until released by either the originating system with an update request or through the user interface. These values are defined in the header file asap_core.h.

Table 2-122 (Cont.) tbl_wo_list

Column_name	Type	Length	Nulls	Description
status	char	1	0	The internal status of the work order to be transmitted to the SARM. Possible values include: • (I) SRP_WO_INIT – Initial state work order. • (H) SRP_WO_HELD – Held work order. • (R) SRP_WO_REVIEW – Reviewed state work order. • (X) SRP_WO_CANCELLED – Cancelled work order. • (E) SRP_WO_TRAN_ERROR – Work order with a translation error. • (F) SRP_WO_FAIL – Failed work order. • (C) SRP_WO_COMP – Completed work order.
due_offset	date	-	0	The due date increment to be applied to the work order when transmitting the work order to the SARM.
parent_wo	char	80	1	If set, the parent work order upon which the work order is dependent. This work order will not begin provisioning until the parent work order has been completed.
batch_group	char	80	1	Batch group for the work order.
extsys_id	varchar2	128	1	Source of the work order (optional).

Table indexes:

```
tst_suite_cd, wo_list_seq_no
```

tbl_wrk_ord (user-created database table)

> **Note:**
>
> This table differs from the table of the same name that is located in the SARM database. See "tbl_wrk_ord (SARM)."

This dynamic table contains instances of logical work orders that belong to a specific test request listed in tbl_tst_rqst. A test request may have one or more work orders.

Table 2-123 tbl_wrk_ord (user-created database table)

Column_name	Type	Length	Nulls	Description
tst_rqst_id	number	38	0	Unique ID of the test request to which the work order belongs.

Table 2-123 (Cont.) tbl_wrk_ord (user-created database table)

Column_name	Type	Length	Nulls	Description
wo_id	char	80	0	Work order ID. The numeric value is the same as wo_unid. This can be useful if some conversion of the work order ID format is required between this database and the work order ID transmitted to SARM.
wo_def_id	char	80	0	Logical work order ID.
asap_stat	char	1	0	Determines the operation field to transmit on the wo_begin RPC from the SRP to the SARM. If HELD_ORDER_STAT (defined in sarm_defs.h) is specified, the work order transmits to the SARM and is held awaiting operator intervention.
estimate	number	38	0	The work order processing estimate returned to the SRP by the SARM after the work order has been transmitted.
revs_flag	char	1	1	The revisions flag returned by the SARM on the work order completion notification.
exceptions	char	1	1	The exceptions flag returned by the SARM on the work order completion notification.

Table indexes:

```
wo_id
```

tbl_wrk_ord_log

This dynamic table contains the information logged for each work order and provides a history of the events that occurred to the work order.

Table 2-124 tbl_wrk_ord_log

Column_name	Type	Length	Nulls	Description
wo_id	char	80	0	Identifies the work order.
event_dt	date	-	0	Date and time of the logged event.
asap_stat	char	8	0	ASAP status of the work order at the time the event occurred. Possible values are detailed for asap_stat in tbl_wo_list.
log_unid	number	38	0	Unique ID of the log wo_log message.
log_msg	varchar2	255	1	A description of the event.

Table indexes:

```
wo_id, log_unid
```

tbl_wrk_ord_parm

This dynamic table contains work order parameters.

Table 2-125 tbl_wrk_ord_parm

Column_name	Type	Length	Nulls	Nulls
wo_id	char	80	0	Unique work order ID to which the parameters belong.
parm_grp	varchar2	80	0	The parameter group associated with the work order ID.
csdl_cmd	char	80	0	The CSDL associated with the work order parameters.
csdl_seq_no	number	38	0	The sequence number of the CSDL within the SRQ.
parm_lbl	varchar2	80	0	Work order parameter label, for example DN, LEN, etc.
parm_vlu	varchar2	255	0	Work order parameter value associated with the work order parameter label.
event_dt	date	-	0	Date and time of the logged event.
parm_identity	number	20	0	Unique ID of the parameter. This is an identity field that is automatically generated by the RDBMS upon insertion.

Table indexes:

```
wo_id, parm_lbl, parm_grp, parm_identity
```

tbl_wrk_ord_rev

This dynamic table contains work order revision information.

Table 2-126 tbl_wrk_ord_rev

Column_name	Type	Length	Nulls	Description
wo_id	char	80	0	Work order ID. The numeric value is the same as wo_unid. This is useful if some conversion of the work order ID format is required between the database and the work order ID transmitted to the SARM.
flag	char	1	0	The revisions flag that is set to (Y)es or (N)o.
csdl_cmd	char	80	0	The CSDL associated with the work order parameters.
parm_seq_no	number	38	0	Sequence number of the CSDL parameter.
parm_lbl	varchar2	80	0	Work order parameter label, for example DN, LEN, etc.

Table 2-126 (Cont.) tbl_wrk_ord_rev

Column_name	Type	Length	Nulls	Description
parm_vlu	varchar2	255	0	CSDL parameter value associated with the parameter label.
parm_desc	varchar2	80	1	Description of the parameter.

Table indexes:

```
wo_id, csdl_cmd, parm_seq_no
```

3
Shared Libraries

This chapter contains information on the ASAP libraries that are shared by the provisioning (upstream) interface and the downstream interface. This chapter consists of the following sections:

Common library interface

This section describes the structures (data types) and functions that are common to both client and server applications. These structures and functions reside in either libasc, or both libclient and libcontrol. If a function is declared in both libclient and libcontrol, its behavior will depend on the implementation specifics in each library.

For example, the API function, ASC_sleep(), is common to both clients and servers, but it works differently in each case. In clients, the function puts the entire process to sleep, but in servers, it only puts the invoking thread to sleep.

The following subsections provide explanations of the variables, prefixes, and concepts used in the functions and data types contained in this section.

Global variables

The following list contains the global variables used in the Common Library Interface.

- **ProgramName:** The current application UNIX executable.
- **ApplName:** The logical name of the current application. To obtain the real application name, in other words, the name that is known to the network through the Interfaces file, refer to ASAP.cfg.
- **ControlSrvName:** The logical name of the Control Server monitoring this application.
- **ApplUserId:** The user ID to log in to the database associated with the application.
- **ApplPassword:** The password for the application.

Open client library API functions

Structures and functions that begin with the following prefixes are part of the Open Client Library:

- **CM_:** Data types and functions common to both clients and servers.
- **CP_:** Data types and functions related to the CLIENT_PROC structure.
- **CPP_:** Data types and functions related to the maintenance of pools of CLIENT_PROC structures used in application servers. This prefix is not used in the common API.

Structures and functions included in the Open Client category are:

- ASC_cpalloc()
- ASC_cpcheck()
- ASC_cpclose()
- ASC_cpfree()
- ASC_cpopen()
- ASC_cprpcexec()
- CLIENT_HANDLER
- CM_RPC
- CM_RPC_PARAM

Oracle Functions

Functions that begin with ASC_oci are part of the Oracle Library. Functions included in this category are:

- ASC_lda_to_oci8
- ASC_oci8_to_lda
- ASC_ocican_cursor
- ASC_ociclose
- ASC_ociclose_cursor
- ASC_ocicreate_cmd
- ASC_ocicreate_list
- ASC_ocidestroy_list
- ASC_ocifetch
- ASC_ociopen
- ASC_ociopen_cursor
- ASC_ociparse
- ASC_ocistatus

I/O management

The structures and functions included in this category are:

- ASC_accept
- ASC_close
- ASC_connect
- ASC_disconnect
- ASC_get_dest_server
- ASC_getc
- ASC_listen
- ASC_open
- ASC_putc
- ASC_read
- ASC_set_fd_blocking
- ASC_set_fd_nonblocking
- ASC_reset_file_status
- ASC_write

Event notification and diagnostic functions

ASAP provides event notification diagnostic functions that applications can use. These functions are described in this chapter.

The system events that may be generated by the ASAP core are described in the “ASAP Core System Events and Alarms" section, which explains how to configure these events to generate system alarms where appropriate.

Structures and functions included in this category are:

- ASC_diag()
- ASC_diag_format
- ASC_diag_on
- ASC_event()
- ASC_event_initialize
- ASC_hex_dump()
- ASC_hex_dump_to_file()
- DIAG_LEVEL

Application configuration determination functions

ASAP includes a facility to configure the entire system and each application separately, by means of name\value parameter pairs in ASAP.cfg. Such parameters are termed configuration parameters within ASAP.

You can retrieve configuration parameter values using ASC_get_config_param().

Memory management functions

The memory management functions are available to both clients and servers within the ASAP environment. Such functions must be used by all clients and servers.

If the application's diagnostic level is KERNEL, these functions log any memory allocation and deallocation to the application's diagnostic log file. Such KERNEL level diagnostics should only be used in the ASAP API.

Structures and functions included in this category are:

- ASC_alloc()
- ASC_bmove()
- ASC_bzero
- ASC_free()
- ASC_realloc()

Performance parameter management

Several performance parameter management functions can be used within ASAP.

Performance parameters are maintained in memory. For application servers, you can retrieve performance parameters from the server with the ASC_dump_param administrative RPC to the particular server.

You may also determine whether you require such performance parameters to be logged to the Control database. If so, set the appropriate configuration parameter for the ASAP Control server.

Self-balancing trees

The self-balancing tree functions are used to manage high-performance self-balancing trees.

Structures and functions included in this category are:

- ASC_create_SBT()
- ASC_delete_element_SBT()
- ASC_delete_index_SBT()
- ASC_destroy_SBT()
- ASC_find_first_SBT()
- ASC_find_free_SBT()
- ASC_find_index_SBT()
- ASC_find_init_SBT()
- ASC_find_next_SBT()
- ASC_insert_element_SBT()
- ASC_walk_SBT()

Date conversion functions

Date conversion functions include the following structures and functions:

- ASC_cur_dts()
- ASC_cur_tm()
- ASC_gettimeofday
- ASC_sec_to_dBdts()
- TODAY()

Miscellaneous functions

The following miscellaneous functions are common to both clients and servers:

- ASC_rstrcmp()
- ASC_sleep()
- ASC_srv_sleep
- get_name_value()

Self-balancing tree examples

This section lists the self-balancing tree examples.

Comparison function

This function defines the logical ordering of elements in the tree.

Syntax:

```
int (*compare_fnt)(void *, void *)
```

Arguments:

- **First:** A void pointer to the first element.
- **Second:** A void pointer to the second element.

Return values:

- -1: The first element is less than the second element.
- 0: The first element is equal to the second element.
- 1: The first element is greater than the second element.

Example:

```
int compare(void *e1, void *e2)
{

int c1=*(int *)e1;
int c2=*(int *)e2;
return ((c1==c2)?0:((c1<c2)?-1:1))

}
```

Delete function

The self-balancing tree element can contain complex data structures. This function deletes an element from the tree and frees the memory used by that element.

Syntax:

```
int (*delete_fnt)(void *)
```

Arguments:

- **First:** A void pointer to the element to be deleted.

Example:

```
int delete(void *e1)
{

ASC_free(e1)

}
```

Action function

This function defines an action that will occur at each element of the tree.

Syntax:

```
int (*action_fnt)(void **, void *, int, int)
```

Arguments:

- **First:** A void pointer to the element to be processed.
- **Second:** A void pointer to the data argument passed to the ASC_walk_SBT function.
- **Third:** An integer defining the level that the element is on. The root element(s) are on level zero.
- **Fourth:** An integer defining the order which the element was scanned. Possible values for the fourth argument are:
  - **INORDER:** Scans all elements in the node then proceeds to the next node.
  - **PREORDER:** Scans the element and then proceeds to the next node.
  - **POSTORDER:** Proceeds to the next node and scans the element on the way back.

Return values:

- 0: If you need to continue to walk the rest of the tree.
- 1: If you are finished with the walk.

Example:

```
int action(void **e1, void *data, int level, int order)
{
if(order==PREORDER)
```

```
{
printf("Found(%d) on level(%d) in order(%d) with data(%d)\n",
**(int **)e1, level, order, *(int *)data);
}

return 0;
}
```

Condition function

Defines the search criteria for the find and delete functions.

Syntax:

```
int (*condition_fnt)(void *, void *)
```

Arguments:

- **First:** A void pointer to the element to determine if it meets the search criteria.
- **Second:** A void pointer to the data argument passed to the ASC_walk_SBT function.

Return values:

- 0: The element does not meet the search criteria.
- 1: The element meets the search criteria.

Example:

```
int condition(void *e1, void *data)
{

return(*(int *)e1==*(int *)data)

}
```

Inline functions

The following list identifies the macros in the Common Library Interface.

- ASC_GET_CMD (CLIENT_PROC *cp): Returns a pointer to the command structure for the connection associated with the client process (for instance, CLIENT_PROC *cp).
- ASC_GET_NAME_SBT(SBT *sbt): Name of SBT (given when created) – used for mutex creation names, etc.
- ASC_GET_SERVER(CLIENT_PROC *cp): Returns the name of the server associated with the client process.
- ASC_GET_USERDATA_SBT(SBT *sbt): Void pointer on SBT. This is used for USER data storage common to SBT.
- ASC_IS_EMPTY_SBT(SBT *sbt): Possible values:
  - 1 – SBT is empty
  - 0 – SBT is not empty
- ASC_IS_OPEN (CLIENT_PROC *cp): Boolean value. Indicates whether the connection is open to the server.
- ASC_NUM_ELEMENTS_SBT(SBT *sbt): Number of elements in SBT.

- TODAY(): The current day since 1970. This macro can be used to determine if the day has changed since last invocation. Used in diagnostic file logging, for instance.
- CM_RPC_PARAM(pinfo): The number of CM_RPC_PARAM records in the table represented by pinfo.

Common library interface functions

This section describes the functions in the Common Interface Library. The functions in this section are listed in alphabetical order.

appl_initialize

This function is the entry point for both ASAP client and server applications. It is called by the relevant API and supplies application-specific logic to the application.

See also appl_cleanup() in client applications.

For more information, see "Multithreaded procedural server initialization."

Syntax:

```
CS_RETCODE appl_initialize(int argc, char *argv[])
```

Arguments:

- **argc:** Number of arguments passed to ASAP on startup.
- **argv:** Array of character pointers to the arguments themselves.

Return values:

- **CS_SUCCEED:** Application initialization was successful.
- **CS_FAIL:** Application initialization failed.

ASC_accept

This function invokes ASC_poll() to determine whether a socket is readable and invokes accept() to accept the next client connection request. The file I/O status of the client socket connection is set to be blocking/nonblocking.

The function returns the new socket stream ID associated with the client connection. It is the caller's responsibility to set I/O control options on the socket file descriptor associated with the client connection.

Syntax:

```
CS_INT ASC_accept (CS_SMALLINT sa_family, CS_INT listen_sockfd, CS_FLOAT
accept_timeout)
```

Arguments:

- **sa_family:** Specifies address family.

  Valid values are: UNIX_ADDR_FAMILY and INET_ADDR_FAMILY (IMPLINK_ADDR_FAMILY and XEROX_NS_ADDR_FAMILY are not supported.)
- **listen_sockfd:** Specifies the listener socket file descriptor ID.

- **accept_timeout:** Specifies the timeout value of the client connect request.

Return values:

- **clt_sockfd:** Socket file descriptor ID for the newly accepted connection.

ASC_alloc

This function allocates the memory on the heap and null pads the allocated memory. If it is unable to do so, it issues a diagnostic message and terminates the server.

This function then initializes the allocated memory before returning a pointer to the appropriate memory segment.

See also ASC_free(), ASC_diag().

Syntax:

```
VOIDPTR ASC_alloc(CS_INT size)
```

Arguments:

- **size:** The size, in bytes, of the memory segment to be allocated.

Return values:

- **ptr:** Void pointer to the allocated initialized memory segment.

ASC_bmove

This function performs a binary data transfer from the source location to the target location in both clients and servers.

See also srv_bmove(), memcpy().

Syntax:

```
void ASC_bmove(CS_VOID *source, CS_VOID *target, CS_INT len)
```

Arguments:

- **source:** Pointer to the source buffer location.
- **target:** Pointer to the target location.
- **len:** Length of the buffer in bytes.

ASC_bzero

This function offers a common interface to null pad memory areas to both servers and clients.

Syntax:

```
void ASC_bzero(CS_VOID *ptr, CS_INT len)
```

Arguments:

- **ptr:** Pointer to the memory to be null padded.
- **len:** Length to be padded.

ASC_close

This function allows threads to close a UNIX device.

Syntax:

```
CS_INT ASC_close(CS_INT dev_fd)
```

Arguments:

- **dev_fd:** UNIX file descriptor being closed.

Return values:

- 0: Device closed successfully.
- -1: Device did not close successfully.

ASC_connect

This function provides server threads and application clients with the same functionality as the UNIX connect function.

Syntax:

```
CS_INT ASC_connect(CS_CHAR socket_client, CS_SMALLINT sa_family, CS_CHAR
*host_name, CS_CHAR *host_ipaddr, CS_USHORT port)
```

Arguments:

- **socket_client:** Specifies whether to connect as socket client or socket server. Valid values are SOCKET_CLIENT, SOCKET_SERVER.
- **sa_family:** The address family. Valid values are UNIX_ADDR_FAMILY and INET_ADDR_FAMILY (IMPLINK_ADDR_FAMILY and XEROX_NS_ADDR_FAMILY are currently unsupported.)
- **host_name:** The host name. If connecting to a UNIX domain socket, this argument specifies socket file path.
- **host_ipaddr:** The host IP address.
- **port:** The service port number.

Return values:

- **0**: Connection to host was successful.
- **-1**: Connection failed.

ASC_convert_msg

This function applies to the correct variable substitution into a specified international message. The message strings are standard C/C++ format strings and are stored in the SARM's database table “tbl_msg_convert". Because the calling function may accept a variable number of arguments (ellipsis notation “..."), there are two methods of accepting data into this function. If the calling function accepts a variable number of arguments, then it must create a va_list and populate it correctly prior to calling this function. If the calling function does not accept a variable number of arguments, then it

simply passes NULL to this function and makes use of the ellipsis provided in this function.

See also ASC_convert_msg_user, ASC_imsg_types(), ASC_imsg_types_user(), ASC_load_msg_tbl().

Syntax:

```
CS_BOOL ASC_convert_msg(CS_CHAR *mbuf, CS_INT bsize, long msg_id, va_list ap, ...)
```

Arguments:

- **mbuf:** Message buffer (of at least size ASC_IMBUF_L) to populate.
- **bsize:** Size of the buffer (ASC_IMBUF_L is recommended as the minimum size) (input) long.
- **msg_id** ID of the message to post (input).
- **va_list_ap:** Variable event parameters for the message (NULL or nothing is sufficient for the messages of type ASC_IMES_NOSUBST_T) (input).

Return values:

- **CS_TRUE:** The function was successfully executed.
- **CS_FALSE:** An error occurred.

Example:

```
void sample_with_va_list(CS_INT msg_id, ...)
{

/*
** The dot dot dot above will be processed and
passed as argument into ASC_convert_msg.
** As developers, we do not know how many
arguments are passed.
*/
va_list ap;
CS_CHAR receive_buffer[MAX_BUFFER_SIZE];
va_start(ap,msg_id);
ASC_msg_convert(receive_buffer , MAX_BUFFER_SIZE ,
msg_id , ap);
va_end(ap);
ASC_diag(NULL , LOW_LEVEL , "sample with va list" ,
__LINE__ , __FILE__ , "returned buffer =
[%s]",receive_buffer);
return;

}
void sample_without_va_list(CS_INT msg_id, CS_INT param1, CS_CHAR *param2, CS_REAL
param3, CS_CHAR param4)
{

/*
** The parameters above (param1, param2, param3,
param4) are fixed and must be defined.  They will
be passed
** directly into ASC_convert_msg
*/
CS_CHAR receive_buffer[MAX_BUFFER_SIZE];
ASC_msg_convert(receive_buffer , MAX_BUFFER_SIZE ,
msg_id , NULL, param1, param2, param3, param4);
ASC_diag(NULL , LOW_LEVEL , "sample without va
```

```
list" , __LINE__ , __FILE__ , "returned buffer =
[%s]",receive_buffer);

return;
```

ASC_convert_msg_user

This function is similar to ASC_convert_msg function, except that programmers can define their own message table.

See also ASC_convert_msg, ASC_imsg_types(), ASC_imsg_types_user(), ASC_load_msg_tbl().

Syntax:

```
CS_BOOL ASC_convert_msg_user(SRQ_MSG_TBL *user_msg_tbl, CS_INT
user_msg_tbl_count, CS_CHAR *mbuf, CS_INT bsize, long msg_id, ...)
```

Arguments:

- **user_msg_tbl:** List of messages contents to be expanded (input).
- **user_msg_tbl_count:** Size of messages contents list (input).
- **mbuf:** Message buffer (of at least size ASC_IMBUF_L) to populate (output).
- **bsize:** Size of the buffer (ASC_IMBUF_L is recommended as the minimum size) (input) long.
- **msg_id:** ID of the message to post (input).

Return values:

- **CS_TRUE:** The function was successfully executed.
- **CS_FALSE:** An error occurred.

ASC_cpalloc

This function allocates and initializes a CLIENT_PROC structure. This function does not open connections to the server.

The configuration parameter USE_GLOBAL_CONTEXT determines whether the global context is used or a separate context is allocated for each connection.

See also ASC_cpopen, ASC_cpfree.

Syntax:

```
CLIENT_PROC *ASC_cpalloc(CS_CHAR *srv_name, CS_CHAR *userid, CS_CHAR *password)
```

Arguments:

- **srv_name:** Name of the server to be used in the connection.
- **userid:** User ID to establish the connection to the server.
- **password:** Password to establish the connection to the server.

  Return values:

  – **NULL:** The function failed and the client process structure could not be allocated.

  - **CLIENT_PROC** *: Pointer to the CLIENT_PROC structure that was allocated and initialized.

ASC_cpcheck

This function checks the network connection to verify that the server is available.

Syntax:

```
CS_RETCODE ASC_cpcheck(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the client process structure that is managing the connection.

Return values:

- **CS_SUCCEED:** Server connection is established and available for processing.
- **CS_FAIL:** The server connection is not available.

ASC_cpclose

This function closes the connection associated with the client process.

See also ASC_cpopen, ASC_cpfree.

Syntax:

```
CS_RETCODE ASC_cpclose(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the client process structure that is managing the connection.

Return values:

- **CS_SUCCEED:** Connection to the server was successfully closed.
- **CS_FAIL:** The function failed and the server connection was not closed.

ASC_cpfree

This function disconnects from the server and frees the client process structure (if applicable).

See also ASC_cpalloc, ASC_cpclose.

Syntax:

```
void ASC_cpfree(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Client process structure allocated with ASC_cpalloc.

ASC_cpopen

This function opens a connection to the server that is specified by the client process structure.

See also ASC_cpalloc, ASC_cpclose.

Syntax:

```
CS_RETCODE ASC_cpopen(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the client process structure that is managing the connection.

Return values:

- **CS_SUCCEED:** Connection to the server has been established.
- **CS_FAIL:** The function failed and the server connection was not created.

When an open server is terminated for any reason, the connections to and from that server are marked as dead (triggers the connection to be closed upon the next use of the connection).

The data structure does not have to be freed and reallocated, because a call to **ASC_CPOPEN** re-establishes the connection that was marked as dead or closed. It is recommended that you explicitly close the connection with **ASC_CPCLOSE** before opening the connection again.

ASC_cprpcexec

This function executes an RPC or registered procedure on the server and processes the results. The results returned by the server are mapped and then the appropriate result handler is called to process the data. If no handler is specified, a default handler provided by the API is executed.

For an example of how to use ASC_cprpcexec, see appl_init.c in $ASAP_BASE/samples.

> **Note:**
>
> $ASAP_BASE refers to the ASAP base path that the current ASAP instance works on.

Syntax:

```
CS_RETCODE ASC_cprpcexec(CLIENT_PROC *cp, CS_VOID *data, CLIENT_HANDLER
*hand_tbl, CM_RPC *rpcdef, ...)
```

Arguments:

- **cp:** Pointer to the client process structure that manages the connection.
- **data:** Generic data pointer that is passed to the handlers once the results are returned by the server.
- **hand_tbl:** Result handler table for processing the return data.
- **rpcdef:** RPC definition structure specifying the RPC to be executed and its associated parameters.
- **...:** Variable parameter list specifying the values for the RPC parameters identified by **rpcdef**. All parameters must be pointers.

Return values:

- **CS_SUCCEED:** RPC execution was successful and the results were handled successfully.
- **CS_FAIL:** The function failed due to an RPC execution error or one of the result handlers failed.

ASC_create_SBT

This function allocates and initializes a self-balancing tree.

Syntax:

```
SBT *ASC_create_SBT(char name, int unique, int (*compare_fnt)(void *, void *))
```

Arguments:

- **name:** Name of the tree or mutex.
- **unique:** Variable to determine whether or not all elements in the self-balancing tree should be unique.
- Valid values are **1** (all elements unique) and **0** (duplicate elements permitted)
- **compare_fnt:** Function that defines the logical ordering of the elements. Valid arguments are **First** (void pointer to the first element) and **Second** (void pointer to the second element).

Return values:

- **NULL:** An error has occurred and the tree was not created.
- **pointer to SBT:** The tree was successfully created.
- **-1:** If the first is less than the second element.
- **0:** If the first is equal to the second element.
- **1:** If the first is greater than the second element.

ASC_cur_dts

This function formats the current date and time and inserts the appropriate fields in the **CS_DATETIME** structure passed to it.

Syntax:

```
void ASC_cur_dts(CS_DATETIME *dts)
```

Arguments:

- **dts:** Pointer to the CS_DATETIME structure that will have its fields updated.

ASC_cur_tm

Return the current time in seconds and microseconds since 00:00 Universal Coordinated Time, Jan 1, 1970. It uses the UNIX function call gettimeofday() to get the current time.

Syntax:

```
CS_FLOAT ASC_cur_tm(void)
```

Return values:

- **Current time:** Current time in seconds and milliseconds in floating point representation.

ASC_delete_element_SBT

This function finds and deletes an element in the tree. Deleting elements from this tree does not affect the balance of the tree.

See also ASC_delete_index_SBT().

Syntax:

```
int ASC_delete_element_SBT(SBT *root, void *datac, int (*condition_fnt)(void *,
void *), void *datad, int (*delete_fnt)(void **, void *))
```

Arguments:

- **root:** Pointer to the tree.
- **datac:** A void pointer that is passed as the second argument to the condition function.
- **condition_fnt:** A function defining the criteria to delete an element. The first element found (in any logical order) will be used. Only one element is deleted per function call.

  Arguments:

  – **First:** Void pointer to the element determines if it meets the search criteria

  – **Second:** Void pointer to the data argument passed to the **ASC_walk_SBT** function

  Return values:

  – **0:** Element does not meet the search criteria

  – **1:** Element meets the search criteria
- **datad:** A void pointer that is passed as the second argument to the delete function.
- **delete_fnt:** Delete function which deletes data allocated prior to the insert; for example, if the element is an entire data structure such as a linked list.

Return values:

- 0: No element matched the criteria or the tree is empty.
- 1: An element was deleted.

ASC_delete_index_SBT()

This function deletes an element from a self-balancing tree (SBT) using the logical ordering of the tree to quickly find the element to delete. The advantage of this delete function over ASC_delete_element_SBT is that it uses the logical ordering and comparison function of the tree to delete an element. This function is therefore more efficient than ASC_delete_element_SBT.

Deleting elements from the tree does not affect the balance of the tree.

Syntax

```
int ASC_delete_index_SBT(SBT *root, void *element, void *data, int (*delete_fnt)(void
**, void *))
```

Arguments

- **root:** Pointer to an SBT. This is the tree to delete the element from.
- **element:** Search node that is used to find the correct element in the SBT to delete. You must populate all of the fields used by the comparison function when the elements are first inserted into the tree.
- **data:** Pointer that is passed into the delete function. Pass NULL if it is not needed.
- **delete_fn:** Function pointer used to delete the element from the tree. You must correctly free up all memory that was allocated for this element (such as a link list or another type of data structure.)

ASC_destroy_SBT

This function deallocates and frees memory used by a self-balancing tree. It is not necessary to delete the elements of the tree prior to destroying the tree.

Syntax:

```
void ASC_destroy_SBT(SBT **root, void *data, int (*delete_fnt)(void **, void *))
```

Arguments:

- **root:** Pointer to the tree.
- **data:** A void pointer that is passed as the second argument to the delete function.
- **delete_fnt:** Function defining the delete algorithm for each element of the tree. This is useful when the element is an entire data structure such as a link-list or another self-balancing tree.

ASC_diag

This function lets you specify a printf() format diagnostic message to be appended to the application's diagnostic file. You may also specify the diagnostic level of the message to be appended to the diagnostic file. If the application's configured diagnostic level is less than or equal to that of the function call, the message will be appended to the diagnostic file. For example, a LOW_LEVEL message will not be written if the server is at SANITY_LEVEL.

See also ASC_event.

You can also call this function with different diagnostic levels throughout the code to denote varying degrees of message logging.

Whenever a PROGRAM_LEVEL or FATAL_LEVEL diagnostic message is logged, the libcontrol ASC_diag() function appends a copy of the message to the application's logfile as well.

The libclient ASC_diag() function writes no message to the client application logfile. The ASC_diag() function is designed for messages that do not exceed the maximum size of 1024 bytes. To display a longer message, use the ASC_diag_format function. This function allows you to customize the presentation of any message (for example, a dump of an incoming message).

The output of the diagnostic file is:

```
>> 141152:12 : LOW:Routing Table:127: router.c
Internal Routing Table: Locked Mutex, Id [15]
```

The fields in the diagnostic file are:

- **>>:** Line identifier.
- **141152:** The time format in hours, minutes, and seconds (hhmmss).
- **12:** The SPID of the thread making the diagnostic function call.
- **LOW:** The level of the diagnostic message.
- **Routing Table:** The type character buffer specified in the diagnostic function. This generally indicates the current function being performed.
- **127:** The line number in the file at which the log entry was generated.
- **router.c:** The file from which the log entry was made.
- **Internal...:** The diagnostic message itself to a maximum diagnostic file size of 3MB. Once the diagnostic file size reaches this limit, the existing file is moved to file.old, and a new diagnostics file started. If this happens again, the original diagnostics file is overwritten. Use the configuration variable, MAX_DIAG_FILE, to specify the application's configuration file that overrides the default size.

For information on configuring diagnostics for Java-based components, see com.mslv.activation.server.Diagnostic in the *ASAP Online Reference*.

Syntax:

```
void ASC_diag(VOIDPTR ptr, DIAG_LEVEL level,

const char *type,
int line,
const char *file,
const char *fmt, ...);
```

Arguments:

- **ptr:** VOIDPTR – only used by the debugger within the Interpreter.
- **level:** The diagnostic level of this particular function call. The possible diagnostic levels are outlined in the enumerated data type DIAG_LEVEL description. See "DIAG_LEVEL abstract data type" for more information.
- **type:** Character pointer identifying the type or functional origin of the function call. Only the first 15 characters of this function call appear in the diagnostic file.
- **line:** The line number of the function in the source file, "__LINE__".
- **file:** The file from which the function was called, "__FILE__".
- **fmt:** Character pointer to a sprintf() format buffer containing the diagnostic message itself.
- **"...":** Variable number of parameters following the spintf() format.

Example:

The following code segment is an example of how ASC_diag() can be used by the ASAP Client and/or Server Application for diagnostic purposes.

```
#include "client.h"
CS_INT SRP_get_wo_revs(ASAP_WO_ID wo_id, ...)
{
CS_INT num_rows = -1;
char *diag_buf = "SRP_get_wo_revs";
/* do some processing here */
...;
ASC_diag(NULL, LOW_LEVEL, diag_buf, __LINE__, __FILE__,
"WO: %s, Successfully Retrieved the WO Revisions", wo_id);
return num_rows;
}
```

ASC_diag_format

You can use this function to generate messages in the diagnostic file of the application. When the function is called, the diagnostic message heading is generated in the log. The specified function is called providing access to the FILE pointer for the diagnostic file. The called function can then generate any message in the diagnostic file.

See also ASC_event.

Syntax:

```
void ASC_diag_format(VOIDPTR ptr,

DIAG_LEVEL level,
const char *type,
int line,
const char *file,
void ((*format_fn)(FILE *diag_outfile, VOIDPTR ptr)));
```

Arguments:

- **ptr:** Pointer to a data segment that is passed into the format_fn as the second argument to allow ASC_diag format to pass information to the format_fn.
- **level:** The diagnostic level of this particular function call. The possible diagnostic levels are outlined in the enumerated data type DIAG_LEVEL description.
- **type:** Character pointer identifying the type or functional origin of the function call. Only the first 15 characters of this function call appear in the diagnostic file.
- **line:** The line number of the function in the source file, "__LINE__".
- **file:** The file from which the function was called, "__FILE__".
- **format_fn:** Pointer to the function that is executed to generate a message in the diagnostic file. When the format_fn function is called, it cannot call any other diagnostic functions.

ASC_diag_on

This function is used to determine whether or not the application is currently generating diagnostic messages at the specified level.

Syntax:

```
CS_BOOL ASC_diag_on(DIAG_LEVEL level)
```

Arguments:

- **level:** The diagnostic level to query for.

ASC_disconnect

This function closes the specified socket connection to the host.

Syntax:

```
CS_RETCODE ASC_disconnect(CS_INT sock_fd)
```

Arguments:

- **sock_fd:** UNIX socket file descriptor.

ASC_dts_to_str

This function converts CS_DATETIME to string datetime format. This function is used primarily to convert datetime formats to strings for insertion into Oracle datetime fields.

Syntax:

```
ASC_dts_to_str(CS_CHAR * date_str, CS_DATETIME *dts)
```

Arguments:

- **date_str:** Pointer to datetime string in the following format: YYYYMMDD 24HH:MM:SS. For example, 20030111 16:04:00. This is the format that the Oracle Server is configured to use and is defined in the set_session database function at login.
- **dts:** Returned ASAP datetime.

ASC_event

This function converts the parameters into a system event request and saves the event in the Control database. Depending on the event type, a system alarm may be generated.

Message size is limited to 80 bytes.

A system alarm can only be generated as a result of a system event. Therefore, to generate alarms, the corresponding events must be generated. If SYS_TERM is the event type, the application terminates.

For information on configuring event generation for Java-based components, see com.mslv.activation.server.EventLog in the *ASAP Online Reference*.

Syntax:

```
void ASC_event(const char *event_type,

short line,
const char *file,
const char *fmt, ...);
```

Arguments:

- **event_type:** Specifies the system event to be generated. This code is used to determine the operation to perform from the configuration tables.

- **line:** Line in source file where system event was generated ("__LINE__").
- **file:** Source file where system event was generated ("__FILE__").
- **fmt:** "printf" type format string specifying the cause of the system event.
- **"...":** Variable number of parameters following the sprintf() format.

ASC_event_initialize

This function initializes system events to be stored in the database. Whether this function saves events to the tbl_event_log is determined by the configuration variable DB_EVENT_LOGGING in the ASAP.cfg file.

Possible values for this variable are:

- **1** – Save events to tbl_event_log.
- **0** – Does not save events to tbl_event_log.

If the events are not saved to the database, then alarms cannot be triggered to execute when these events are generated.

Syntax:

```
CS_RETCODE ASC_event_initialize (void)
```

ASC_find_first_SBT

This function finds the first element that meets the criteria specified in the ASC_find_init_SBT function. The element that is returned can be accessed directly to change the data. It is recommended that the key fields be left unchanged so that the logical ordering of the tree is not disrupted.

Syntax:

```
int ASC_find_first_SBT(SBT_FIND *head,

SBT_FIND **current,
void **element)
```

Arguments:

- **head:** Pointer to results of the previous ASC_findinit_SBT function. No maintenance is required with this data structure.
- **current:** Pointer that scans the found records. No maintenance is required with this data structure.
- **element:** A void pointer that receives the first found element.

Return values:

- **0:** No elements were found.
- **1:** An element was found.

ASC_find_free_SBT

This function deallocates and frees the memory used by the find functions.

The **current** variable used in the ASC_find_first_SBT and ASC_find_next_SBT functions is not valid after this function is called and should therefore be set to NULL.

Syntax:

```
void ASC_find_free_SBT(SBT_FIND **head)
```

Arguments:

- **head:** Pointer to the results of previous ASC_findinit_SBT function. No maintenance is required with this data structure.

ASC_find_index_SBT

This function returns an element that matches the search record using the logical ordering of the tree as an index. This function uses the same comparison function as ASC_insert_element_SBT to traverse the tree. To use this function, the programmer must ensure that the unique argument is set for the tree (as defined in ASC_create_index_SBT).

The reason for the uniqueness requirement is that only one record is returned (the first to meet the comparison criteria). A simple way to guarantee the uniqueness of the index field(s) is to set the unique flag when creating the tree and when inserting a record:

```
if ASC_insert_element_SBT!= NULL then

update record

endif
```

Syntax:

```
void *ASC_find_index_SBT(SBT *root,

void *data)
```

Arguments:

- **root:** Pointer to the tree to walk.
- **data:** A void pointer that contains the data structure that is stored in the tree with the indexed field(s) containing the data to find in the tree.

Return values:

- **0:** No element was found.
- **void *:** Pointer to the indexed element that was found.

ASC_find_init_SBT

This function allocates and initializes the find function. This function performs the search on the tree via a tree walk algorithm. To take advantage of the logical ordering of the tree, use ASC_find_index_SBT. Observe the following programming technique:

```
if ASC_find_init_SBT = 1 then

while ASC_find_next_SBT = 1
process found record
ASC_find_free
```

```
else

no record was found

endif
```

Syntax:

```
int ASC_find_init_SBT(SBT *root,

SBT_FIND **head,
void *data,
int(*condition_fnt)(void*, void*))
```

Arguments:

- **root:** Pointer to the tree to walk.
- **head:** Pointer that receives the results of the search. The first time that this function is used this variable MUST be set to NULL. Before scanning the tree, ASC_find_init_SBT deletes the previous find results and then passes it to ASC_find_first_SBT and ASC_find_next_SBT.
- **data:** Void pointer that is passed as the second argument to the condition function.
- **condition_fnt:** Function that defines the search criteria for the elements.

Return values:

- **0:** No elements were found.
- **1:** An element was found.

ASC_find_next_SBT

This function finds the next element that meets the criteria specified in the ASC_find_init_SBT function. You do not need to call ASC_find_first_SBT prior to this function.

Syntax:

```
int ASC_find_next_SBT(SBT_FIND *head,

SBT_FIND **current,
void **element)
```

Arguments:

- **head:** Pointer to results of previous ASC_findinit_SBT function. No maintenance is required with this data structure.
- **current:** Pointer that scans the found records. No maintenance is required with this data structure.
- **element:** Void pointer that receives the next found element.

Return values:

- **0:** No elements were found.
- **1:** An element was found.

ASC_free

This function deallocates the memory previously allocated using ASC_alloc(). If it is unable to do so, it issues a system event and terminates the server.

See also ASC_alloc(), ASC_diag().

Syntax:

```
void ASC_free(VOIDPTR ptr)
```

Arguments:

- **ptr:** Void pointer to a memory segment previously allocated with **ASC_alloc()**.

ASC_GET_CMD

This is a pointer to the command in the current context on the client process structure.

Syntax:

```
ASC_GET_CMD(cp)
```

ASC_get_config_param

This function references the ASAP.cfg configuration file and returns the value of the requested parameter.

If this type of entry is not present as either an application or global configuration parameter, then the default value is returned.

The configuration variables listed in the configuration file are read initially by the application and stored internally from then on. Any change to the configuration files requires the application to be restarted in order to use the new settings.

Syntax:

```
CS_RETCODE ASC_get_config_param(char *param,

char *value,
char *default_val)
```

Arguments:

- **param:** The parameter name itself.
- **value:** Character pointer to the value of the requested parameter. The returned parameter value is placed in this location.
- **default_val:** Character pointer to the default value to be used for this parameter if the parameter is not listed in the configuration file.

Return values:

- **CS_SUCCEED:** Parameter value found in the configuration file or the default value returned.

ASC_GET_CONTEXT

This is a pointer to the context on the client process structure.

Syntax:

```
ASC_GET_CONTEXT(cp)
```

ASC_GET_SERVER

This is a pointer to the server name on the client process structure.

Syntax:

```
ASC_GET_SERVER(cp)
```

ASC_getc

Gets characters in a non-blocking mode from a stream file. Read timeout is achieved in UNIX code by setting alarm() before calling the getc system call.

The alarm() call cannot be used to time out the call in the ASAP server because the Sleep Manager thread provided by the ASAP Server API uses alarm() to implement sleep management. ASC_getc() calls ASC_poll() to provide timeout. For uniformity, the same approach is followed in the Client API.

See also ASC_poll(), ASC_putc(), UNIX getc(), putc().

The function declaration is the same as for the UNIX getc except for the additional parameter that specifies the timeout in seconds.

This function assumes that the file descriptor has not been set to nonblocking mode.

This function invokes ASC_poll() to check if port is readable. It is not intended to be used to read from disk files.

Syntax:

```
CS_INT ASC_getc(FILE *stream,

CS_FLOAT timeout)
```

Arguments:

- **stream:** Open file descriptor to read from.
- **timeout:** Timeout in seconds.

Return values:

- **c:** The character read/written to the stream file.
- **-1:** A UNIX error occurred on getc/putc.
- **-2:** ASC_read/ASC_write timed out.
- **-3:** Port hangup detected.
- **-4:** EOF returned by getc/putc due to a UNIX error and no port hangup.

ASC_gettimeofday

This function gives the API a consistent method of calling the gettimeofday() UNIX function.

Syntax:

```
ASC_gettimeofday(tp, tzp)
```

ASC_hex_dump

By specifying a buffer, length, and diagnostic level similar to ASC_diag(), this function provides a method of generating a hexadecimal dump of the selected buffer in the applications diagnostic logfile.

See also ASC_diag().

Syntax:

```
void ASC_hex_dump(DIAG_LEVEL level,

const char *type,
int line,
const char *file,
CS_BYTE *buf,
int buf_len);
```

Arguments:

- **level:** The diagnostic level of the function call. See ASC_diag() for more information.
- **type:** Brief description of the circumstances of the function call to identify such entries in the applications logfile.
- **line:** The line in the source file at which the function was called.
- **file:** The source file from which this function was called.
- **buf:** The character buffer for which the hexadecimal dump is to be generated.
- **buf_len:** The length of the buffer.

ASC_hex_dump_to_file

This function writes the specified binary buffer to a file in hexadecimal format.

Syntax:

```
void ASC_hex_dump_to_file(FILE *fptr,

CS_BINARY *buf,
int len)
```

Arguments:

- **fptr:** File pointer to open destination file for the hex dump.
- **buf:** Buffer containing the data to be dumped.
- **len:** Length of the buffer in bytes.

ASC_imsg_types

This function determines the formatting mode and destination types of the specified international message.

See also ASC_load_msg_tbl(), ASC_convert_msg().

Syntax:

```
CS_BOOL ASC_imsg_types(long msg_id,

CS_CHAR *frm_type,
CS_CHAR *dest_type)
```

Arguments:

- **long msg_id:** ID of the message to post (input).
- **frm_type:** Formatting type (one character long) of the message (may be NULL) (output).

  Substitution type can be one of the following: ASC_IMES_NOSUBST_T, ASC_IMES_SUBST_T, CS_CHAR
- **Dest_type:** Destination type (one character long) of the message (may be NULL) (output).

  The destination type can be one of the following: ASAP_LOG_SRQ, ASAP_LOG_WOA, ASAP_LOG_SRQWOA, ASAP_LOG_NO

Return values:

- **CS_TRUE:** On success.
- **CS_FALSE:** On failure.

ASC_imsg_types_user

This function determines the formatting mode and destination types of the specified international message from a user-defined message table.

See also ASC_load_msg_tbl(), ASC_convert_msg().

Syntax:

```
CS_BOOL ASC_imsg_types_user(SRQ_MSG_TBL *user_msg_tbl,

CS_INT user_msg_tbl_count,
long msg_id,
CS_CHAR *frm_type,
CS_CHAR *dest_type)
```

Arguments:

- **user_msg_tbl;** List of messages contents to be expanded (input).
- **user_msg_tbl_count:** Size of messages contents list (input).
- **long msg_id:** ID of the message to post (input).
- **frm_type:** Formatting type (one character long) of the message (may be NULL) (output).

  Substitution type can be one of the following: ASC_IMES_NOSUBST_T, ASC_IMES_SUBST_T, CS_CHAR

- **Dest_type:** Destination type (one character long) of the message (may be NULL) (output).

  The destination type can be one of the following: ASAP_LOG_SRQ, ASAP_LOG_WOA, ASAP_LOG_SRQWOA, ASAP_LOG_NO.

Return values:

- **CS_TRUE:** The function was successfully executed.
- **CS_FALSE:** An error occurred.

ASC_insert_element_SBT

This function inserts an element into the tree. Inserting elements into the tree does not affect the balance of the tree. Therefore, a resort to randomness algorithm is not required on the key fields to improve on the balancing of the tree, as is the case with a binary search tree.

Syntax:

```
void *ASC_insert_element_SBT(SBT *root,

void *element)
```

Arguments:

- **root:** Pointer to the tree to insert into.
- **element:** A void pointer to the element to be inserted. It is the caller's responsibility to maintain the memory for the element since only a pointer is stored in the tree.

Return values:

- **NULL:** The node is inserted into the tree.
- **not NULL:** Pointer to an existing node that matches the key fields of the node to be inserted. No new node was inserted.

ASC_IS_OPEN

This is a boolean flag to determine whether the current connection is open on the client process structure.

Syntax:

```
ASC_IS_OPEN(cp)
```

ASC_lda_to_oci8

This function switches the Oracle connection between oci8 and oci7.

After obtaining a CLIENT_PROC, using ASC_cppalloc() for example, the ASC_oci8_to_lda bridge API must be called to switch the OCI8 context to OCI7 LDA. Once this is done, the OCI7 wrapping functions (such as ASC_ociopen_cursor(), ASC_ociparse(), ASC_ocirpcexec() and so on) can be used.

Similarly, before freeing a CLIENT_PROC using, for example, ASC_cppfree(cp), the ASC_lda_to_oci8 API must be called to switch OCI7 LDA back to OCI8 context.

If using a high level API like ASC_cprpcexec(), this switching mechanism is not required.

Several high level wrapping functions support one connection-to-cursor array relationships. As a result, ASC_ociopen() does not open a cursor. The function to open a cursor has been moved to ASC_oci8_to_lda(). Without this bridge, the OCI parse call will fail.

Syntax:

```
CS_RETCODE ASC_oci8_to_lda( CLIENT_PROC *cp)
```

Arguments:

- **cp:** Connection to the SARM database.

Return values:

- **CS_SUCCEED:** Successfully switched an Oracle connection.
- **CS_FAIL:** Failed to switch an Oracle connection.

ASC_listen

This function listens for connection requests on the specified port by invoking the appropriate local function based on the socket address family you specify.

The caller can then invoke ASC_accept() to accept the next incoming client connection request.

Setting socket options is not currently supported by ASC_connect().

Syntax:

```
CS_INT ASC_listen (CS_SMALLINT sa_family,

CS_CHAR *host_name,
CS_CHAR *host_ipaddr,
CS_USHORT port,
CS_INT backlog)
```

Arguments:

- **sa_family:** Specifies the address family. Valid values are: UNIX_ADDR_FAMILY, INET_ADDR_FAMILY. (IMPLINK_ADDR_FAMILY, and XEROX_NS_ADDR_FAMILY are not supported)
- **host_name:** The host name. If it is a UNIX domain socket, it specifies the socket file path.
- **host_ipaddr:** The host IP address.
- **port:** The service port number.
- **backlog:** The maximum allowable length of the queue for pending connections. If a connection request arrives when the queue is full, the client receives an error.

  Backlog is currently limited by the system (silently) to be in the range of 1 to 20. If any other value is specified, the system automatically assigns the closest value within range.

Return values:

- **Listener Socket FID:** The socket file descriptor ID that listens for incoming client requests is returned if the call succeeds.
- **-1:** If the call fails.

ASC_load_msg_tbl

This function loads international messages and audit destinations from the SARM database into memory. It executes the function SSP_load_msg_tbl using the requested language code.

Syntax:

```
CS_BOOL ASC_load_msg_tbl(CLIENT_PROC *cp,

CS_CHAR *lang)
```

Arguments:

- **cp:** Connection to the SARM database.
- **lang:** Language code of messages to be loaded. If NULL, it uses the value of the configuration parameter LANGUAGE_OF_MSG. If the configuration parameter is not defined, it defaults to “USA".

Return values:

- **CS_TRUE:** The function was successfully executed.
- **CS_FALSE:** An error occurred.

ASC_oci8_to_lda

This function switches the Oracle connection between oci7 and oci8.

Syntax:

```
CS_RETCODE ASC_oci8_to_lda( CLIENT_PROC *cp)
```

Arguments:

- **cp:** Connection to the SARM database.

Return values:

- **CS_SUCCEED:** Successfully switched an Oracle connection.
- **CS_FAIL:** Failed to switch an Oracle connection.

ASC_ocican_cursor

This function cancels a query on a cursor after the desired rows have been fetched.

This function is used to free up resources after you have completed processing the required number of rows and there are rows still pending in the result set.

Syntax:

```
CS_RETCODE ASC_ocican_cursor( Cda_Def  *cda)
```

Arguments:

- **cda:** Pointer to a cursor data area structure.

Return values:

- **CS_SUCCEED:** The function was successfully executed.
- **CS_FAIL:** The function failed.

ASC_ociclose

This function closes a connection between an Open Server or Open Client and an Oracle database (using oci8 calls).

Syntax:

```
CS_RETCODE ASC_ociclose(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the CLIENT_PROC structure to be closed.

Return values:

- **CS_SUCCEED:** Successfully closed an Oracle connection.
- **CS_FAIL:** Failed to close an Oracle connection.

ASC_ociclose_cursor

This function closes a cursor. It disconnects a cursor from the data areas in the Oracle Server with which it is associated.

Syntax:

```
CS_RETCODE ASC_ociclose_cursor(Cda_def *cda)
```

Arguments:

- **cda:** Pointer to cursor data area structure.

Return values:

- **CS_SUCCEED:** The function closed the cursor.
- **CS_FAIL:** The function failed to close the cursor.

ASC_ocicreate_cmd

This function builds a PL/SQL block from the passed CM_RPC structure. The resulting command is passed to ASC_ociparse to be associated with an open cursor.

Syntax:

```
CS_RETCODE ASC_ocicreate_cmd(CS_CHAR *command, CM_RPC *rpcdef)
```

Arguments:

- **command:** PL/SQL command buffer.
- **rpcdef:** CM_RPC structure containing RPC name and associated parameters.

Return values:

- **CS_SUCCEED:** The function successfully built PL/SQL block.
- **CS_FAIL:** The function to build the PL/SQL block.

To receive a return status from an RPC to the Oracle server, functions are used on the server side instead of procedures. Therefore, all command strings receive a return value.

ASC_ocicreate_list

This function defines an output variable for each column of the result set. This function uses the **odescr** function to determine the column name, data type, and data size for every column in the cursor variable. After the column information is determined, it is used to populate the call to **odefin**. The odefin function is required to define the storage array for each column in the result set.

Syntax:

```
CS_INT ASC_ocicreate_list(CLIENT_PROC *cp, Cda_Def *cda, ORA_COLUMN *colsptr,
CS_INT numrows)
```

Arguments:

- **cp:** Points to a CLIENT_PROC structure.
- **cda:** Points to a CDA structure.
- **colsptr:** Points to an Oracle column structure that is defined while processing the resultant data.
- **numrows:** The number of rows to allocate in the ORA_COLUMN structure. This parameter governs how many rows are returned with each fetch of the result set.

Return values:

- **CS_INT:** The number of columns described in the result set.

ASC_ocidestroy_list

This function deallocates all memory allocated during the ASC_ocicreate_list function. ASC_ocidestroy_list must accompany each use of ASC_ocicreate_list after processing is complete. If not, these resources will not be available.

Syntax:

```
CS_RETCODE ASC_ocidestroy_list(ORA_COLUMN *cols)
```

Arguments:

- **cols:** Points to an Oracle column structure that was defined while processing the resultant data.

Return Value:

- **CS_SUCCEED:** The function was successful.
- **CS_FAIL:** The function failed.

ASC_ocifetch

This function attempts to fetch as many rows as were defined by the numrows argument of the function ASC_ocicreate_list.

Syntax:

```
CS_RETCODE ASC_ocifetch(CLIENT_PROC *cp, Cda_Def *cda, ORA_COLUMN * cols)
```

Arguments:

- **cp:** Points to a CLIENT_PROC structure.
- **cda:** Points to a CDA structure.
- **cols:** Points to an Oracle column structure that was defined while processing the resultant data.

Return values:

- **CS_SUCCEED:** Row returned successfully.
- **CS_FAIL:** No more rows to process.

ASC_ociopen

Opens a single Oracle connection using oci8 calls.

ASC_ociopen() does not open a cursor. The function to open a cursor has been moved to ASC_oci8_to_lda(). Custom code must explicitly call ASC_oci8_to_lda() before parsing. Without this bridge, the OCI parse call will fail.

Syntax:

```
CS_RETCODE ASC_ociopen(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the CLIENT_PROC structure to be opened

Return values:

- **CS_SUCCEED:** Successfully opened an Oracle connection.
- **CS_FAIL:** Failed to open an Oracle connection.

ASC_ociopen_cursor

This function opens a cursor. It associates a cursor data area in the application with a data area in the Oracle server. Cursor data areas are used by Oracle to maintain state information about the processing of a SQL statement.

Syntax:

```
CS_RETCODE ASC_ociopen_cursor( CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to a CLIENT_PROC structure.

Return values:

- **CS_SUCCEED:** The function opened the cursor.
- **CS_FAIL:** The function failed to open the cursor.

ASC_ociparse

This function parses a SQL statement or SQL block and associates it with the cursor data area found in CLIENT_PROC member cda. Note that for performance reasons, all parsing is

performed in deferred mode. This means that any errors in the command are not detected until the command is executed.

Syntax:

```
CS_RETCODE ASC_ociparse(CLIENT_PROC *cp, CS_CHAR *command)
```

Arguments:

- **cp:** Pointer to a CLIENT_PROC structure.
- **command:** PL/SQL command.

Return values:

- **CS_SUCCEED:** The function successfully parsed the PL/SQL block.
- **CS_FAIL:** The function failed to parse the PL/SQL block.

ASC_ocistatus

This function checks the value of the passed CDA to provide error management and diagnostics. In addition, this function is responsible for determining whether the initial OCI call is blocked (due to network, server response etc.) and initiating the synchronous appearance behavior by establishing a poll() on the connection file descriptor found in the CLIENT_PROC member, connection_fd.

Syntax:

```
CS_VOID ASC_ocistatus(CLIENT_PROC *cp, Cda_Def *cda);
```

Arguments:

- **cp:** Pointer to a CLIENT_PROC structure.
- **cda:** Points to a CDA structure.

Return values:

- **CS_SUCCEED:** The function was successful.
- **CS_FAIL:** The function failed.

ASC_open

This function allows threads to open UNIX devices without blocking the application server. It is the thread-level version of the UNIX open system call.

This function is intended to be used to open devices that might potentially block and uses ASC_poll() to check if the port is writable. It is not intended to be used to open disk files.

Syntax:

```
CS_INT ASC_open(CS_CHAR *path_name,

CS_INT flags,
CS_FLOAT open_timeout)
```

Arguments:

- **path_name:** Pathname to the UNIX device being accessed.

- **flags:** Flags controlling the mode that the device is opened with. The flags used here are analogous to those used in the UNIX “open()" system call.
- **open_timeout:** Timeout value, in seconds, to wait for the open call to be successful.

Return values:

- **>=0:** Operation was successful and the UNIX file descriptor is returned.
- **-1:** Operation failed.

ASC_putc

Puts characters in a non-blocking mode to a stream file. Read timeout is achieved in UNIX code by setting alarm() before calling the **putc** system call.

The alarm() call cannot be used to timeout the call in ASAP server since the Sleep Manager thread provided by ASAP Server API uses alarm() to implement sleep management. ASC_putc() calls ASC_poll() to provide timeout. For uniformity, the same approach is followed in the Client API.

See also ASC_poll(), UNIX getc(), putc().

The function declaration is the same as for the UNIX putc except for the additional parameter that specifies timeout in seconds.

Until all bytes are written or ASC_poll times out, ASC_putc loops the ASC_poll() to check if the port is readable/writable or timed out, and then calls read/write.

This function assumes that the file descriptor has not been set to nonblocking mode.

This function invokes ASC_poll() to check if the port is writable. It is not intended to be used to write to disk files.

Syntax:

```
CS_INT ASC_putc(CS_INT c,

FILE *stream,
CS_FLOAT timeout)
```

Arguments:

- **stream:** Open file descriptor to write to.
- **timeout:** Timeout in seconds.

Return values:

- **c:** The character read/wrote to the stream file.
- **-1:** A UNIX error occurred on getc/putc.
- **-2:** ASC_read/ASC_write timed out.
- **-3:** Port hangup detected.
- **-4:** EOF returned by getc/putc due to a UNIX error and no port hangup.

ASC_read

This function provides timeout using nonblocking read. Read timeout is achieved in UNIX code by setting alarm() before calling a read system call. The alarm() call to timeout the read

call cannot be used since the Sleep Manager thread provided by ASAP Server API uses alarm() to implement sleep management. Instead, ASC_read() calls ASC_poll() to provide timeout.

This function assumes that the file descriptor has not been set to nonblocking mode.

This function declaration is the same as for the UNIX read except for the additional parameter that specifies timeout in seconds.

Until all bytes are read or ASC_poll times out, ASC_read loops the ASC_poll() to check if the port is readable or timed out, and then calls ASC_read.

This function is intended to be used to read devices that might potentially block and uses ASC_poll() to check if port is readable. It is not intended to be used to read from disk files.

Syntax:

```
CS_INT ASC_read(CS_INT fd,

void *buf,
CS_INT size
CS_INT *bytes_read,
CS_FLOAT timeout)
```

Arguments:

- **fd:** Open file descriptor to read from or write to.
- **buf:** Data buffer to read to or write from.
- **size:** Size of buffer.
- **bytes_read:** Actual bytes read.
- **timeout:** Number of seconds for read timeout. (Within Open Server, you can obtain granularity less than seconds). Valid values are: -1, 0, >0. If the timeout_seconds parameter value is -1, **ASC_write** blocks until the port is readable. This feature should not be used within the Open Server application since all files are supposed to be opened and set for nonblocking I/O. If timeout is set to 0, **ASC_read** returns immediately and mimics UNIX read().

Return values:

- **0:** Success.
- **-1:** A UNIX error occurred.
- **-2:** ASC_read timed out.
- **-3:** Port hangup detected.
- **-4:** EOF detected.
- **-5:** Network Operational Error.

ASC_realloc

This function initializes newly allocated memory, copies the contents of the previously allocated memory, and frees previously allocated memory before returning a pointer to the reallocated memory segment.

See also ASC_free(), ASC_diag().

If this function is unable to reallocate previously allocated memory on the heap, this function issues a system event and terminates the process. If the size of the reallocated memory segment is specified as zero, the previously allocated memory is available and a null pointer returned.

The content of the previously allocated memory is preserved if the size of the reallocated memory segment is greater than the current size. If not, it is truncated.

Syntax:

```
VOIDPTR ASC_realloc(VOIDPTR ptr,

CS_INT size,
CS_INT cur_size)
```

Arguments:

- **ptr:** Pointer to previously allocated and used memory.
- **size:** The size, in bytes, of the reallocated memory segment.
- **cur_size:** The size, in bytes, of the currently allocated memory segment.

Return values:

- **new_ptr:** Void pointer to the re-allocated (possibly moved) memory segment.
- **NULL:** If requested memory size is zero bytes.

ASC_reset_file_status

This function resets the file status to the previous file status flag value.

For more information on fcntl(), refer to UNIX documentation.

Syntax:

```
CS_RETCODE ASC_reset_file_status(CS_INT fd,

CS_INT oflags)
```

Arguments:

- **fd:** File descriptor value.
- **oflags:** Saved file status flag value.

Return values:

- **CS_SUCCEED:** Operation successful.
- **CS_FAIL:** Operation on the file descriptor failed.

ASC_rstrcmp

This function performs a reverse string comparison. It is used in B Tree search functions to compare nodes in a more random manner to generate a more balanced B tree structure.

Syntax:

```
int ASC_rstrcmp(const char *buf1,

const char *buf2)
```

Arguments:

- **buf1, buf2:** Character pointers to the strings to be compared.

Return values:

- **>0:** First string is lexicographically greater than the second when compared in reverse.
- **<0:** Second string is lexicographically greater than the first when compared in reverse.
- **0:** First string is lexicographically equal to the second.

ASC_sec_to_dBdts

This function converts the UNIX time in seconds to a database format record.

Syntax:

```
void ASC_sec_to_dBdts(time_t sec,

CS_DATETIME *dts)
```

Arguments:

- **sec:** UNIX time in seconds.
- **dts:** Pointer to datetime variable to return the converted time value in.

ASC_set_fd_blocking

This function sets files to blocking mode, saving the current file status flag value. Blocking mode I/O must not be performed from within an Open Server, since it will block the whole server.

For more information on fcntl(), refer to UNIX documentation.

Syntax:

```
CS_RETCODE ASC_set_fd_blocking(CS_INT fd,

CS_INT *flags)
```

Arguments:

- **fd:** File descriptor value.
- **flags:** Integer pointer. Saves current file status flag value.

Return values:

- **CS_SUCCEED:** Operation successful.
- **CS_FAIL:**Operation on the file descriptor failed.

ASC_set_fd_nonblocking

This function sets files to nonblocking mode, saving the current file status flag value.

For more information on fcntl(), refer to UNIX documentation.

Syntax:

```
CS_RETCODE ASC_set_fd_nonblocking(CS_INT fd,

CS_INT *flags)
```

Arguments:

- **fd:** File descriptor value.
- **flags:** Integer pointer to save current file status flag value.

Return values:

- **CS_SUCCEED:** Operation successful.
- **CS_FAIL:** Operation on the file descriptor failed

ASC_set_new_handler

This function sets the _new_handler global variable to point to the new_exeption_hdl() callback function. The previous _new_handler value is not required, and is neither returned nor saved in a static variable.

Both Control API and Client API main() invoke this function when the ASAP server/client application is initialized. This eliminates the new exception handling that is specific to a platform or complier.

Syntax:

```
void ASC_set_new_handler(CS_BOOL diag_initialized)
```

Arguments:

- **diag_initialized:** Boolean variable. Specifies whether ASAP diagnostics and event management variables have been initialized or not. Uses either system calls or ASAP API calls to log messages and terminate the application.

ASC_sleep

This function provides a thread with a sleep function similar to the UNIX sleep() function.

See also ASC_wakeup().

If the application is an application server, this function puts the calling thread to sleep for the specified time interval. If the application is an application client, this function puts the entire client process to sleep for the specified time interval.

Syntax:

```
void ASC_sleep(time_t seconds)
```

Arguments:

- **seconds:** The sleep period, in seconds, before the application is woken up.

ASC_str_to_dts

This function converts the string date time format to ASAP datetime. This function is used primarily to convert datetime strings from Oracle database results to ASAP datetime format.

Syntax:

```
ASC_str_to_dts(CS_CHAR * date_str, CS_DATETIME *dts)
```

Arguments:

- **date_str:** Pointer to datetime string in the following format: YYYYMMDD 24HH:MM:SS. For example, 19980111 10:04:00. This is the format which is returned from the Oracle Server as defined in the set_session database function at login.
- **dts:** Returned ASAP datetime.

ASC_walk_SBT

This function walks the tree and performs an action at each node.

For more information, refer to the Action Function example on "Action function."

Syntax:

```
void ASC_walk_SBT(SBT *root,

void *data,
int(*action_fnt)(void **, void *, int, int))
```

Arguments:

- **root:** Pointer to the tree to walk.
- **data:** A void pointer that is passed as the second argument into the action function.
- **action_fnt:** Function that defines the action performed by the elements. Note that a pointer to a pointer is passed and, therefore, the data can change. Do not edit the key fields as the logical ordering is disrupted.
- **First:** A void pointer to the element to be processed.
- **Second:** A void pointer to the data argument passed to the ASC_walk_SBT function.
- **Third:** An integer defining the level that the element is on. The root element(s) are on level zero.
- **Fourth:** An integer defining the order in which the element was scanned. Possible values are:
  - **INORDER** – Scans all elements in node before proceeding to the next.
  - **PREORDER** – Scans the elements and then proceeds to the next node.
  - **POSTORDER** – Proceeds to the next node and scans the element on the way back.

Return values:

- **0:** You need to continue to walk the rest of the tree.
- **1:** You are finished with the walk.

ASC_write

This function provides a timeout using a nonblocking write. Write Timeout is achieved in UNIX code by setting alarm() before calling a write system call.

This function assumes that the file descriptor has not been set to nonblocking mode.

The alarm() call to time out the write call cannot be used since the Sleep Manager thread provided by ASAP Server API uses alarm() to implement sleep management. Instead, ASC_write() calls ASC_poll() to provide timeout.

This function declaration is the same as the UNIX write function, except for the additional parameter that specifies the timeout in seconds.

Until all bytes are written or ASC_poll times out, ASC_write loops the ASC_poll() call to check if the port is readable/writable or timed out, and then calls write.

This function invokes ASC_poll() to check if port is readable. It is not intended to be used to read from disk files.

Syntax:

```
CS_INT ASC_write(CS_INT fd,

const void *buf,
CS_INT size,
CS_INT *bytes_written,
CS_FLOAT timeout)
```

Arguments:

- **fd:** Open file descriptor to read from or write to.
- **buf:** Data buffer to read from or write to.
- **size:** Size of buffer.
- **bytes_written:** Actual number of bytes written.
- **timeout:** Number of seconds for write timeout. Within the Open Server, you can specify a granularity of less than a second. Valid values are: -1, 0, >0. If the timeout_seconds parameter is a value of -1, ASC_write blocks until the port is writable. This should not be used within the Open Server application since all files are supposed to be opened and set for nonblocking I/O. If timeout is set to 0, ASC_write returns immediately and mimics UNIX write().

Return values:

- **0:** Okay.
- **-1:** A UNIX error occurred.
- **-2:** ASC_write timed out.
- **-3:** Port hangup detected.
- **-5:** Network Operational Error (host down, network down, and so on)

get_name_value

This function extracts name and value parts for the next message line beginning in the start position you specify in the input transaction buffer.

The message is in the format NAME=VALUE;\n and the buffer should look like: NAME=VALUE;\n[NAME=VALUE;\n]. Ensure that name and value buffers are large enough to hold the information.

Syntax:

```
int get_name_value(char *input_buf,

int start_pos,
int buf_len,
char *name,
char *value)
```

Arguments:

- **input_buf:** Character buffer containing the transaction.
- **start_pos:** Specifies start position to get current line. (Input)
- **buf_len:** Specifies length of input buffer. (Input)
- **name:** Variable to save the name as part of the message in line. (Output)
- **value:** Variable to save the value as part of the message in line. (Output)

Return values:

- **int:** Start position for next message line in the input buffer.

MS_DIFF

This function determines the difference between two times expressed in milliseconds.

Syntax:

```
MS_DIFF(start, end)
```

TODAY

This function returns the current day of the year.

Syntax:

```
TODAY()     today()
int today(void)
```

Return values:

- Day of the year: Current day of the year.

Example:

int yday; yday = TODAY();

Common library interface data types

The following list provides an overview of all the data types found in the Common Library Interface.

For detailed descriptions and a list of public members, arguments, return values, and remarks associated with each of data types, refer "Common library interface data types."

CLIENT_HANDLER

Informs API to call certain functions given certain return results.

- **CM_RPC:** Defines an RPC to invoke or defines a registered procedure.

- **CM_RPC_PARAM:** Defines a parameter for an RPC or registered procedure.
- **DIAG_LEVEL:** Specifies the valid diagnostic levels of an application process using the diagnostic API functions.

The following section describes all of the data types in the Common Library Interface. The functions and structures included in this section are listed in alphabetical order.

CLIENT_HANDLER abstract data type

This structure informs the API which functions to call when certain results are returned. If you do not include a return result, the API uses the default processing to handle the data for the result set and continues processing. The default processing generally ignores the result row.

If you want to use the default processing for a return result type, do not include the result type in the table.

Syntax:

```
typedef struct {

CS_RETCODE (*handler)(CLIENT_PROC *cp, CS_VOID *data,
CS_INT res_type, CS_BOOL *not_done)
CS_INT res_type;

} CLIENT_HANDLER;
```

Members:

- **handler:** The handler function to call when a return result is matched. To indicate the end of the table, this field should be set to NULL.
- **res_type:** Return result type. Refer to the description for more information regarding result types.

CM_RPC abstract data type

This structure is used to define an RPC to invoke or define a registered procedure.

Syntax:

```
typedef struct{

CS_CHAR *rpcname;
CM_RPC_PARAM *paraminfo;
CS_INT numparam;
CS_RETCODE (*reghandler) (SRV_PROC *srvproc)

} CM_RPC;
```

Members:

- **rpcname:** Name of RPC or registered procedure.
- **paraminfo:** Pointer to the parameter structure. Set it to NULL if there is no parameter.
- **numparam:** Total number of parameters defined in this structure. Set to NULL if there is no parameter required, otherwise use the macro NUM_RPC_PARAM().
- **reghandler:** The handler function that is called when a registered procedure arrives.

CM_RPC_PARAM abstract data type

This structure is used to define the parameter for RPCs and registered procedures. It is used to retrieve parameters when receiving a registered procedure or sending an RPC. The parameters defined in this structure should be in the correct order expected for sending and receiving. The first parameter is passed to the function as item 1.

Syntax:

```
typedef struct {
    CS_DATAFMT dfmt;
    CS_BYTE *datap;
    CS_INT datalen;
} CM_RPC_PARAM;
```

Arguments:

- **dfmt:** Data description structure defined by the header file which stores necessary format information for the parameter.
- **datap:** Pointer to the data as default parameter used to define a registered procedure or retrieved from incoming registered procedure.
- **datalen:** Specify the length of data pointed by the datap member. This is only used when setting up default data. Check srv_regparam for details.

DIAG_LEVEL abstract data type

This enumeration specifies the valid diagnostic levels of an application process using the diagnostic API functions.

The diagnostic levels are used with the ASC_diag() function and other diagnostic functions within the API.

If the diagnostic level of the process is higher or equal to the diagnostic level of the ASC_diag() function, then that diagnostic message is written to the diagnostic file.

The diagnostic levels are:

- PROG
- SANE
- LOW
- KERN

PROG provides the lowest level of detail of the diagnostics. The details become progressively greater for each level, with KERN being the most detailed.

The remaining levels are only used to provide backwards compatibility for ASAP, and are not available to be set.

Syntax:

```
typedef enum {

KERNEL_LEVEL,
LOW_LEVEL,
FUNCTION_LEVEL,
RPC_LEVEL,
```

```
CONTRACT_LEVEL,
SANITY_LEVEL,
PROGRAM_LEVEL,
FATAL_LEVEL

} DIAG_LEVEL;
```

Members:

- **KERNEL_LEVEL:** Used by the kernel to generate diagnostic messages. It is only to be used by the core libraries for very low-level debugging of core code. You can set the application diagnostic level to KERN.
- **LOW_LEVEL:** Used by the application to generate low-level diagnostic messages from any of its functions. Such messages enable the programmer to debug an application. Once debugged, the diagnostic level of the application should be changed to provide less detail. You can set the application diagnostic level to LOW.
- **FUNCTION_LEVEL:** Used by the application at the beginning and end of each function to track the operation of the application. Not generally used in the core application.
- **RPC_LEVEL:** Used by the application to produce RPC diagnostic messages.
- **CONTRACT_LEVEL:** Used to specify the start and end of a particular instance of a contract.
- **SANITY_LEVEL:** Used by the application for high-level diagnostics. This level of diagnostic messages provides user information about the processing of the system. It is used for low level diagnostic messages. A production application has its diagnostic level set at either PROG or SANE.
- **PROGRAM_LEVEL:** This is primarily used to generate error messages when the application is running in a production environment.
- **FATAL_LEVEL:** Used for fatal error conditions if the process is terminated. Only used if an error condition occurs within the application such that if the application continued, more errors would occur and compound the problem. For instance, if a function is missing from the database, then the application terminates for manual intervention.

Server library interface

This section details the API functionality provided to ASAP application servers. This functionality is provided by the API library, libcontrol.

This section describes server applications you can use to do the following:

- Manage threads.
- Receive registered procedures or remote procedure calls from client applications.
- Handle language requests.
- Manage the creation of pools of client connections.
- Set up pools of DBPROCESS connections to the SQL server.
- Incorporate I/O functions within an application server.
- Set up utility thread functions.
- Set up gateway functionality.

Functions and structures

The following subsections provide you with reference information on the variables, prefixes, and concepts used in the functions and structures contained in the server library interface.

Global variables

This section lists the global variables defined by this server API:

- **RealSrvName:** Actual name of the application server as known to the network (that is, the name in the Interfaces file). To obtain this name from the application configuration file, use the logical application name ApplName as the configuration parameter.
- **server_context:** Global context for the application server.
- **ASAP_high_availability:** Boolean flag. Indicates whether or not High Availability mode is active.
- **ASAP_IS_ALIVE_INTERVAL:**Global polling interval for checking connections between the servers.

Thread management functions

The server library interface provides the following thread management structures and functions:

- ASC_alarm()
- ASC_await_init_completion()
- ASC_lockmutex
- ASC_malarm()
- ASC_msleep()
- ASC_reg_init_func()
- ASC_spawn()
- ASC_unlockmutex
- background_process_init()
- BACKGROUND_PROCESS

Memory management functions

The server library interface provides the following memory management structures and functions:

- ASC_blk_alloc
- ASC_blk_free
- ASC_blk_realloc()

RPCs and registered procedures

This section outlines the steps involved to have the application server receiving registered procedures or remote procedure calls from client applications.

Structures and functions included in this category are:

- add_appl_rpc()
- add_registered_proc()
- add_rpc
- ASC_define_events()
- ASC_define_rpc()
- ASC_get_reg_param()
- ASC_get_rpc_param()
- ASC_handle_results
- REG_PROC
- RPC
- RPC_PARAM
- USEREVENT

Language requests

This section includes functions and structures to help you handle language requests. Structures and functions included in this category are:

- add_lang_handler()
- ASC_convert_msg
- ASC_convert_msg_user
- ASC_createmsgq
- ASC_deletemsgq
- ASC_getmsgq
- ASC_imsg_types
- ASC_imsg_types_user
- ASC_load_msg_tbl
- ASC_putmsgq
- ASC_send_text()
- LANG_HANDLER

Client process connection pool functions

Client process connection pool functions create a pool of client connections to SQL servers. These functions help you manage the creation of pools of client connections for any thread to use within the application server.

The pools of connections assume that connections to the SQL server do not need to be checked periodically to determine whether the server is still running. Connections to other application servers are checked regularly because these servers may become unavailable. If this happens, all connections to that server are released. If communicating with another application server, an application server spawns a driver thread to manage the network connections and check the connection periodically.

Structures and functions included in this category are:

- ASC_cppalloc()
- ASC_cpdbpcreate
- ASC_cppfree()
- ASC_in_system()
- ASC_in_territory
- ASC_cpdbpdestroy

Thread I/O functions

This section includes I/O functions you can incorporate within an application server. Structures and functions included in this category are:

- ASC_poll()
- ASC_poll_timer()

Utility thread functions

Structures and functions included in this category are:

- ASC_getpid()
- ASC_threadproc()
- ASC_sendinfo()
- ASC_thread_field_bool()
- ASC_thread_field_int()
- ASC_thread_field_str()
- ASC_srv_field_bool()
- ASC_srv_field_int()
- ASC_srv_field_str()
- ASC_lock_strtok()
- ASC_stack_trace()
- ASC_unlock_strtok()

Inline functions

The inline functions defined by this server API are as follows:

- **APPL_STATE(state):** Sets the descriptive state for the current thread. It is displayed by the sp_ps and sp_who system procedures.

- **ASC_exit():** Terminates the application server.
- **ASC_SRV_MSGQID:** Open Server message queue ID that was created due to ASC_spawn.
- **ASC_SRV_DATA:** Pointer to the global data segment for the thread.
- **ASC_SRV_START-UP:** Pointer to the startup data segment for the thread.
- **ASC_GET_RPC_PARAM (srvproc, rpc_name, param_name, paramno, dest_type, dest, dest_len, max_len):** Retrieves a parameter from an RPC call. (**Note:** This cannot be used for RPCs or registered procedures that are written using the System 10 model.)

Server application functions

This section details server application functions. These functions and structures are listed in alphabetical order.

add_appl_rpc

This function adds an RPC to the application server, either as an RPC or a registered procedure depending on a configuration parameter.

See also add_rpc, ASC_define_rpc.

This function has been superseded by ASC_define_rpc().

Syntax:

```
CS_RETCODE add_appl_rpc(RPC *rpc)
```

Arguments:

- **rpc:** Pointer to the RPC definition structure.

Return values:

- **CS_SUCCEED:** Addition successful.
- **CS_FAIL:** Too many RPCs added to the server.

add_lang_handler

This function scans the current list of installed language handlers and installs a new language handler at the end of the list. If the maximum number (100) of language handlers has been exceeded, a system event is issued.

Syntax:

```
void add_lang_handler(LANG_HANDLER *hand)
```

Arguments:

- **hand:** Pointer to a language handler structure containing the language handler details.

Example:

```
static int run_lang_handler(SRV_PROC *srvproc, CS_CHAR *langptr, CS_INT langlen);
/* Language Handler Definition */
static LANG_HANDLER run_lang_def = {
    "run", "run", run_lang_handler
};
```

```
void init_function(void)
{
...
    add_lang_handler(&run_lang_def);
}
static int run_lang_handler(SRV_PROC *srvproc, CS_CHAR *langptr, CS_INT langlen)
{
    /* Process the language buffer */
    ...
    /* Complete the language request */
    srv_sendstatus(srvproc, 0);
    srv_senddone(srvproc, SRV_DONE_FINAL, 0, 0);
    return CS_SUCCEED;
}
```

add_registered_proc

This function adds the registered procedure specified by the **rp** argument to the application server.

The use of this function is no longer recommended. Instead, use the ASC_define_rpc() function.

Syntax:

```
CS_RETCODE add_registered_proc(REG_PROC *rp)
```

Arguments:

- **rp:** Pointer to the REG_PROC structure.

Return values:

- **CS_SUCCEED:** Addition successful.
- **CS_FAIL:** Too many RPCs added to the server.

add_rpc

This function adds a procedure to a server as a remote procedure. Use this function to add procedures that take optional parameters as registered procedures do not allow optional arguments.

Only use this function to define RPCs to the server. Such procedures take variable numbers of arguments. If the procedure takes a fixed number of non-null arguments, it should be defined as a registered procedure.

Syntax:

```
CS_RETCODE add_rpc(RPC *rpc)
```

Arguments:

- **rpc:** Pointer to the RPC structure.

Return values:

- **CS_SUCCEED:** Addition successful.
- **CS_FAIL:** Too many RPCs added to the server.

ASC_alarm

This function sets up an asynchronous alarm for a thread. The **msg** argument uniquely identifies the alarm and must point to a data segment where the first four bytes is the message operation. When the alarm expires, the message is sent to msgqid with the operation set to msg_operation. To cancel the alarm, set the seconds argument of this function to zero.

Any thread calling this function MUST be awaiting messages on a previously created message queue.

The thread message structure must have a long integer as the first field. This integer field is used as the message operation field, identifying the message type to the receiving thread. The first action of this function is to assign the msg pointer to the msg_operation value, thus setting the message operation for the alarm wakeup message. After this first message field, the message can have any format that the sending and receiving threads agree upon.

Syntax:

```
void ASC_alarm(time_t seconds, SRV_OBJID msgqid, int *msg, long msg_operation)
```

Arguments:

- **seconds:** Time, in seconds, after which the alarm is issued.
- **msgqid:** The thread message queue to which the thread message is to be sent. Note that the calling thread must have a dedicated thread message queue in order to receive the alarm notification.
- **msg:** An integer pointer to the previously allocated alarm message data area.
- **msg_operation:** The message operation used to identify an alarm message being received.

ASC_await_init_completion

This function waits for the initialization process to complete before continuing to execute.

Syntax:

```
void ASC_await_init_completion(void)
```

ASC_blk_alloc

This function returns a pointer to a memory block of a specified size (bytes). The memory is from a preallocated pool. ASC_BLK_ALLOC is called to interface with this function.

See also ASC_mem_alloc.

Each memory pool has a default number of blocks. This value is configurable in ASAP.cfg.

Syntax:

```
VOIDPTR ASC_blk_alloc(int line, char *file, CS_INT size, CS_CHAR *type_name)
```

Arguments:

- **line:** The line number of the function in the source file, “__LINE__".
- **file:** The file from which the function was called, “__FILE__".

- **size:** The number of bytes required.
- **type_name:** Name of type to be allocated.

ASC_BLK_ALLOC

This macro selects a block of memory from the appropriate memory pool that satisfies the size requirements of the structure specified by **count * struct_name**.

One block of memory is returned that is large enough to hold a number of instances of the structure. The number of instances is specified by **count**.

If required, you can change the number of blocks in each memory pool in ASAP.cfg.

Table 3-1 Default Memory Pool Blocks

Size of Each Block in the Pool	Default Number of Blocks	Configuration Variable
16	1024	ASC_BLOCK16_POOL
32	1024	ASC_BLOCK32_POOL
64	1024	ASC_BLOCK64_POOL
96	1024	ASC_BLOCK96_POOL
128	512	ASC_BLOCK128_POOL
256	256	ASC_BLOCK256_POOL
512	128	ASC_BLOCK512_POOL
1024	64	ASC_BLOCK1024_POOL
2048	32	ASC_BLOCK2048_POOL
4096	16	ASC_BLOCK4096_POOL
8192	8	ASC_BLOCK8192_POOL

Syntax:

```
(struct_name*) ASC_BLK_ALLOC (struct_name, struct_desc, count)
```

Example:

```
void function()
{

MY_STRUCTURE item1, item2;
MY_STRUCTURE *array;
array = ASC_BLK_ALLOC (MY_STRUCTURE, "MY_STRUCTURE", 1);
array[0] = item1;
array = ASC_BLK_REALLOC (array, MY_STRUCTURE, "MY_STRUCTURE", 2);
array[1] = item2;
ASC_BLK_FREE (array);

}
```

ASC_BLK_FREE

This macro returns the block of memory, specified by ptr, to its memory pool. The default number of memory blocks is configured in ASAP.cfg.

For an example of the use of this macro, refer to ASC_BLK_ALLOC.

Syntax:

```
void ASC_BLK_FREE (ptr)
```

ASC_blk_realloc

This function returns a pointer to a new block of a specified size (bytes). The contents of the specified address are moved to the new block. ASC_BLK_REALLOC is called to interface with this function.

For more information on the default number of blocks assigned to memory pools, see "ASC_BLK_ALLOC."

Each memory pool has a default number of blocks. The ASC_BLOCK##_POOL value is configured in ASAP.cfg.

Syntax:

```
VOIDPTR ASC_blk_realloc(VOIDPTR addr, int line, char *file, CS_INT size, CS_CHAR
*type_name)
```

Arguments:

- **addr:** The address of current allocation.
- **line:** The line number of the function in the source file, “__LINE__".
- **file:** The file from which the function was called, “__FILE__".
- **size:** Number of bytes required.
- **type_name:** Name of type to be allocated.

ASC_BLK_REALLOC

This macro selects a block of memory from the appropriate pool that will satisfy the size requirements of the structure specified by **count * struct_name**.

For an example of the use of this macro, refer to ASC_BLK_ALLOC.

Copies the contents of the previous block (specified by ptr) into the new block, and then returns the previous block to its pool.

One block of memory is returned that is large enough to hold a number of instances of the structure. The number of instances is specified by **count**.

The default number of memory blocks is configured in ASAP.cfg.

Syntax:

```
struct_name*) ASC_BLK_REALLOC (ptr, struct_name, struct_desc, count)
```

ASC_cpdbpcreate

This function creates a CLIENT_PROC connection pool to the database server of the size you specify.

Syntax:

```
CLIENT_PROC_POOL ASC_cpdbpcreate(CS_CHAR *pool_name, CS_CHAR *srv_name, CS_CHAR
*userid, CS_CHAR *password, CS_INT size, CP_CONNECT_TYPE cp_type)
```

Arguments:

- **pool_name:** Name of the client process pool.
- **srv_name:** Name of the server to establish a connection with.
- **userid:** User ID for security validation by the server.
- **password:** Password for security validation by the server.
- **size:** Number of client connection processes to be established.
- **cp_type:** Connection type. Valid values include:
  - OPEN_SERVER – Sybase Open Server Connection type.
  - ORACLE – Oracle Server RDBMS Connection type.

ASC_cpdbpdestroy

This function destroys the client process connection pool and frees all processes.

Syntax:

```
void ASC_cpdbpdestroy(CLIENT_PROC_POOL *cpp)
```

Arguments:

- **cpp:** Pointer to the client pool.

ASC_cppalloc

This function allocates a client process from the client process connection pool.

See also ASC_cppfree.

Syntax:

```
CLIENT_PROC *ASC_cppalloc(CLIENT_PROC_POOL *cpp)
```

Arguments:

- **cpp:** Pointer to the client process pool.

Return values:

- **NULL:** The function failed and the client process could not be allocated. This will only happen if the client process pool is invalid.
- **CLIENT_PROC*:** Pointer to the allocated client process.

ASC_cppfree

This function frees an allocated client process and returns it to the original pool.

See also ASC_cppalloc.

Syntax:

```
void ASC_cppfree(CLIENT_PROC *cp)
```

Arguments:

- **cp:** Pointer to the client process structure.

ASC_createmsgq

This function creates an Open Server message queue. When a message queue is created, a mutex for that queue is also created. Therefore, the Open Server must be configured to allow the creation of enough mutexes.

See also ASC_deletemsgq, ASC_putmsgq and ASC_getmsgq.

Syntax:

```
CS_RETCODE ASC_createmsgq(CS_CHAR *mqname,

CS_INT mqlen,
SRV_OBJID *mqid)
```

Arguments:

- **mqname:** Pointer to the message queue name.
- **mqlen:** Length of the message queue name.
- **mqid:** Pointer to the message queue ID.

Return values:

- **CS_SUCCEED:** The message queue was created successfully.
- **CS_FAIL:** The creation of the message queue failed.

ASC_createmutex

This function creates a mutex.

See also ASC_deletemutex, ASC_lockmutex, and ASC_unlockmutex.

Syntax:

```
CS_RETCODE ASC_createmutex(CS_CHAR *mutex_name,

CS_INT mutex_namelen,
SRV_OBJID *mutex_id)
```

Arguments:

- **mutex_name:** The name of the mutex.
- **mutex_namelen:** The length of the mutex name if not null terminated.
- **mutex_id:** The mutex ID returned during creation.

Return values:

- **CS_SUCCEED:** The mutex was created successfully.
- **CS_FAIL:** An error in the operation.

ASC_define_events

This function installs handlers for user-defined events.

Syntax:

```
CS_RETCODE ASC_define_events(USEREVENT *tbl)
```

Arguments:

- **tbl:** Table of user events to be installed into the application server.

Return values:

- **CS_SUCCEED:** The user events were successfully installed into the application server.
- **CS_FAIL:** The installation failed.

ASC_define_rpc

This function registers a procedure with the application server.

Syntax:

```
CS_RETCODE ASC_define_rpc(CM_RPC *reg_def)
```

Arguments:

- **reg_def:** Pointer to the RPC definition structure.

Return values:

- **CS_SUCCEED:** Addition successful.
- **CS_FAIL:** RPC could not be defined in the server.

ASC_deletemsgq

This function deletes an Open Server message queue. The mutex created when the message queue was created will be removed.

See also ASC_createmsgq, ASC_putmsgq, and ASC_getmsgq.

Syntax:

```
CS_RETCODE ASC_deletemsgq(CS_CHAR *name,

CS_INT length,
SRV_OBJID id)
```

Arguments:

- **name:** Pointer to the name of the message queue.
- **length:** Length of the message queue.
- **id:** The ID of the message queue.

Return values:

- **CS_SUCCEED:** The message queue was deleted successfully.
- **CS_FAIL:** The deletion of the message queue failed.

ASC_deletemutex

This function deletes a mutex.

See also ASC_createmutex, ASC_lockmutex, and ASC_unlockmutex.

Syntax:

```
CS_RETCODE ASC_deletemutex(CS_CHAR *mutex_name,

CS_INT mutex_namelen,
SRV_OBJID mutex_id)
```

Arguments:

- **mutex_name:** The name of the mutex.
- **mutex_namelen:** The length of the mutex name if not null terminated.
- **mutex_id:** The mutex ID specified during deletion.

Return values:

- **CS_SUCCEED:** The mutex was deleted successfully.
- **CS_FAIL:** An error occurred during the operation.

ASC_get_reg_param

This function determines the parameters for a registered procedure.

Syntax:

```
CS_RETCODE ASC_get_reg_param(SRV_PROC *srvproc, CM_RPC *reg_def, ...)
```

Arguments:

- **srvproc:** Pointer to the current thread structure.
- **reg_def:** Pointer to the registered procedure definition.
- **...:** Variable list of data value pointers.

Return values:

- **CS_SUCCEED:** Addition successful.
- **CS_FAIL:** The determination of the parameters on the registered procedure failed.

ASC_getmsgq

This function retrieves a thread message.

See also ASC_createmsgq, ASC_putmsgq and ASC_deletemsgq.

Syntax:

```
CS_RETCODE ASC_getmsgq(SRV_OBJID msgqid,

CS_VOID **msg,
CS_INT flags,
CS_INT *info)
```

Arguments:

- **msgqid:** Name of the message queue.
- **msg:** Indirect pointer to the thread message.
- **flags:** Informational flag – Populated upon failure.
- **info:** Information – Populated upon failure.

Return values:

- **CS_SUCCEED:** The message was received successfully.
- **CS_FAIL:** The message was not received.

ASC_getpid

This function gets the server process ID for the thread associated with the current server process.

Syntax:

```
CS_INT ASC_getpid(SRV_PROC *srvproc)
```

Arguments:

- **CS_INT:** Open Server process ID for the server process you specified.
- **srvproc:** The Open Server process handle.

Return values:

- **SRV_PROC***: Pointer to current server process.
- **CS_INT:** Open Server process ID for the server process you specified.

ASC_get_securedata

Retrieves a secure data entry.

Syntax:

```
CS_RETCODE ASC_get_securedata(char *name,

char*value);
```

Arguments:

- **name:** Used as key to retrieve secure data entry.
- **value:** Encrypted password field of the secure data entry.

ASC_handle_results

Once a command has been sent to the SQL Server, this function processes the results and passes them back to the Open Server client.

Syntax:

```
CS_RETCODE ASC_handle_results(CLIENT_PROC *rmtproc,
```

```
SRV_PROC *srvproc,
CS_INT *last_row_cnt,
CS_INT *tot_row_cnt)
```

Arguments:

- **rmtproc:** The Open Client Library handle to the remote DBMS.
- **srvproc:** The Open Server process handle to use to send results to the client.
- **last_row_cnt:** Pointer to the number of rows transmitted to the client in the last set of results in the command batch. This can be used by the calling function to determine whether any such rows were returned to the client and to set the row count in the final srv_senddone() statement. If not required, set this field to NULL.
- **tot_row_cnt:** Pointer to the total number of rows transmitted to the client. This determines whether any command result rows were transmitted to the client. If not required, set this field to NULL.

Return values:

- **CS_SUCCEED:** Successfully returned data rows (if any).
- **CS_FAIL:** Error.

ASC_in_system

This function checks whether or not the specified component is defined in the specified territory and system. This function is related to high-availability installations.

Syntax:

```
CS_RETCODE ASC_in_system(CS_CHAR *territory,

CS_CHAR *system,
CS_CHAR *component)
```

Arguments:

- **territory:** Name of the ASAP territory to check.
- **system:** Name of the ASAP system to check within the territory.
- **component:** Name of the component to check in the system.

Return values:

- **CS_SUCCEED:** The component is defined for the ASAP territory and system.
- **CS_FAIL:** The component is not defined for the ASAP territory and system.

ASC_in_territory

This function checks if the specified component is defined in the specified territory.

Syntax:

```
CS_RETCODE ASC_in_territory(CS_CHAR *territory,

CS_CHAR *component)
```

Arguments:

- **territory:** Name of the ASAP territory to check.
- **component:** Name of the component to check in the system.

Return values:

- **CS_SUCCEED:** The component is defined for the ASAP territory.
- **CS_FAIL:** The component is not defined for the ASAP territory.

ASC_lockmutex

This function locks mutexes.

See also ASC_unlockmutex.

Syntax:

```
CS_RETCODE ASC_lockmutex(SRV_OBJID mutex_id,

CS_INT waitflag,
CS_INT *infop)
```

Arguments:

- **mutex_id:** The mutex ID returned during locking.
- **waitflag:** Specifies whether the thread requesting the lock of the mutex should wait or return if the mutex cannot be locked.
- **infop:** A pointer to a CS_INT. Refer to the appropriate Sybase documentation for appropriate values.

Return values:

- **CS_SUCCEED:** The mutex was locked successfully.
- **CS_FAIL:** An error in the operation.

ASC_lock_strtok

Since the UNIX function strtok() maintains a global or static variable of its current position within the string, it is not multi-thread-safe. ASC_lock_strtok is used in conjunction with strtok() in a multi-threaded environment. After you have finished using strtok(), ASC_unlock_strtok() must be called to free the associated mutex.

Syntax:

```
void ASC_lock_strtok(void)
```

ASC_malarm

This function sets up an asynchronous alarm for a thread. It provides the same functionality as ASC_alarm but is used to provide timing of less than one second.

See also "ASC_alarm."

Syntax:

```
void ASC_malarm(CS_FLOAT  timeout,
```

```
SRV_OBJID msgqid,
int *msg,
long msg_operation)
```

Arguments:

- **timeout:** Time, in milliseconds, after which the alarm goes off.
- **msgqid:** The thread message queue to which to send the thread message. Note that the calling thread must have a dedicated thread-message queue in order to receive the alarm notification.
- **msg:** An integer pointer to the previously allocated alarm message data area.
- **msg_operation:** The message operation used to identify an alarm message being received.

ASC_mem_alloc

This function allocates a fixed size block from the memory pool.

See also ASC_mem_free, ASC's libcontrol RPC call “mem_usage" for usage statistics, and ASC_mem_create libcontrol API call to create a memory pool.

Upon startup, a **pool** of memory is created. The pool can hold several records of information. If the pool is not large enough, it is automatically resized to accommodate the new records. When ASC_mem_alloc is called, a record is assigned as in_use and returned as a VOIDPTR (a generic C pointer which can be defined as any type).

The ASAP Memory Manager controls the constant allocation, deallocation, and reallocation of memory. These procedures are expensive in terms of CPU usage. The pool is allocated only once. All the ASC_mem_allocs and ASC_mem_free functions that are called, simply update the in_use flag of each memory record.

Syntax:

```
VOIDPTR ASC_mem_alloc(int line, char *file,

ASC_MEM_POOL *pool)
```

Arguments:

- **line:** The line number of the function in the source file, "__LINE__".
- **file:** The file from which the function was called, "__FILE__".
- **ASC_MEM_POOL *pool:** The name of the memory pool.

ASC_mem_free

This function frees a fixed sized block and returns it to the memory pool.

See also ASC_mem_alloc, ASC's libcontrol RPC call “mem_usage" for usage statistics, and ASC_mem_create libcontrol API call to create a memory pool.

When the ASC_mem_alloc pointer allocated is no longer needed, ASC_mem_free automatically returns it to the correct pool.

The ASAP Memory Manager controls the constant allocation, deallocation, and reallocation of memory. These procedures are expensive in terms of CPU usage. The pool is allocated

only once. All the ASC_mem_allocs and ASC_mem_free functions that are called update the in_use flag of each memory record.

Syntax:

```
void ASC_mem_free(int line,

char *file,
VOIDPTR p)
```

Arguments:

- **line:** The line number of the function in the source file, "__LINE__".
- **file:** The file from which the function was called, "__FILE__".
- **VOIDPTR:** A generic C pointer which can be defined as any type.

ASC_msleep

This function allows a thread to sleep for the time, in seconds, that you specify. If you want to set the time in milliseconds, specify the fractional part of a second.

See also ASC_sleep.

Syntax:

```
void ASC_msleep(CS_FLOAT seconds)
```

Arguments:

- **seconds:** Number of seconds to send the thread into sleep state.

ASC_poll

This function is the Application server poller that provides timeout. The Open Server Poller function srv_poll() in the blocking mode (SRV_M_WAIT) does not provide a timeout. The alarm() call cannot be used to time out the srv_poll() call since the Sleep Manager thread provided by ASAP Server API uses the alarm() to implement sleep management. ASC_poll() provides srv_poll() functionality allowing for timeout like UNIX I/O multiplexing call poll().

This function declaration is the same as for srv_poll() except for the additional parameter that specifies the timeout in seconds. There is also an additional return value, -2, which indicates that srv_poll() timed out.

ASC_poll() implements the timeout by creating a socketpair and sending the write endpoint of the socketpair in a timeout request message to the sleep manager. The read endpoint of the socketpair is added to the list of ports to be watched for. If an event occurs in any of the ports, the sleep manager is sent a timer cancellation, and the return value from the srv_poll() is returned. Otherwise, if the timeout message from sleep manager is received on the read endpoint of the socketpair connection, it returns a value of -2 to indicate timeout.

All ASAP Application servers should call ASC_poll() instead of directly calling srv_poll().

Setting a timeout of 0 is equivalent to calling srv_poll() with wait-flag set to SRV_M_NOWAIT.

Setting a timeout of -1 is equivalent to calling srv_poll() with wait-flag set to SRV_M_WAIT.

Setting a timeout > 0 provides poll-like functionality.

Syntax:

```
CS_INT ASC_poll(SRV_POLLFD *fdsp,

CS_INT nfds,
CS_FLOAT timeout)
```

Arguments:

- **fdsp:** Pointer to an array of SRV_POLLFD structures with one element for each open file descriptor of interest.
- **nfds:** Number of elements in the ***fdsp** array.
- **timeout:** Number of seconds for srv_poll timeout. Open servers do not support granularity of less than seconds. Valid values are: -1, 0, > 0. If the timeout_seconds parameter is a value of -1, ASC_poll() does not return until at least one specified event has occurred. If the value of the parameter is 0, ASC_poll() does not wait for an event to occur but returns immediately, even if no specified event has occurred. This is consistent with UNIX poll behavior.

Return values:

- **n:** The number of file descriptions.
- **0:** No file descriptors are ready, or ASC_poll timed out. This is consistent with UNIX poll behavior.
- **-1:** An error occurred.

ASC_poll_timer

This function sets up an asynchronous alarm to provide timeout while polling for data for a thread. The **msg** argument uniquely identifies the timer and must point to a data segment where the first (4 bytes) is the message operation. When the timer expires, the message integer pointer is written to the socket sock_fid.

See also "ASC_poll ."

To cancel the timer, set the timeout argument to zero (0.0) seconds.

Any thread calling this function MUST be polling for messages on the read end of the socketpair connection.

The format and contents of the message may have any format because the **msg** pointer is written to the sock_fid. The caller must be responsible for handling if the message pointed to has been deallocated.

Syntax:

```
void ASC_poll_timer(CS_FLOAT timeout,

int sock_fid,
int *msg)
```

Arguments:

- **timeout:** Time, in seconds, specified as the floating point value after which the timer expires. The granularity is to the millisecond.
- **sock_fid:** Write end of the socketpair connection over which the timer notification is sent. The calling thread must create a socketpair, allocate one end as the write end and the other as the read end, send it to sleep manager as part of ASC_poll_timer(), and poll for messages on the read end of the socketpair.
- **msg:** An integer pointer to the previously allocated timer message data area. The caller is responsible for populating the data area with information needed on timeout notification from the sleep manager and deallocating the data area.

ASC_putmsgq

This function adds a thread message to a particular message queue.

See also ASC_createmsgq, ASC_getmsgq, and ASC_deletemsgq.

Syntax:

```
CS_RETCODE ASC_putmsgq(SRV_OBJID mqid,

CS_VOID *msgp,
CS_INT flags)
```

Arguments:

- **mqid:** Name of the message queue.
- **msgp:** Pointer to the thread message.
- **flags:** Message processing flags.

Return values:

- **CS_SUCCEED:** The message was sent successfully.
- **CS_FAIL:** An error occurred during message send.

ASC_reg_init_func

This function registers an initialization function within the application server with a priority specified by its first argument. Such functions execute in serial order by priority by the initialization thread within the server API. While this thread executes these initialization functions, an initialization mutex is locked, preventing other threads from starting up until all the initialization functions have finished.

Such initialization functions can perform network I/O activities such as loading static tables from the database, etc. This type of network activity cannot be performed from within the SRV_START handler. Therefore, it is necessary to spawn a separate initialization thread to execute these tasks.

The initialization thread does not begin executing until the SRV_START event handler has finished, that is, appl_initialize() has returned, allowing the application to register its initialization functions before the initialization thread begins.

This function must be called from within the SRV_START handler function, either in the API or in the appl_initialize() application-supplied function.

Syntax:

```
void ASC_reg_init_func(int level,

char *desc,
CS_RETCODE (*init_func)(void))
```

Arguments:

- **level:** Priority of the initialization function. 0 to 10 is reserved for the API. The application server code can use > 10.
- **desc:** Description of the initialization function.
- **init_func:** Function pointer to initialization function.

ASC_send_text

This function sends a language command, text buffer, and if required, the contents of the specified file to an application server. The command that is placed on the first line of the text to be sent determines which language handler the destination server invokes.

See also add_lang_handler().

If the filename is set, the command, the contents of the file, and the buffer are transmitted in that order. If the filename is NULL, the command and buffer are transmitted.

This function also determines whether or not there is a thread already spawned to act as a language driver thread to the specified application server. If a thread exists, it sends a synchronous thread message to that thread, which then transmits the text buffer and returns a status value to the calling function.

If no such thread currently exists, a language driver thread is spawned within the server process to establish and maintain a network connection to the destination application server. From this point on, all text transmission requests are directed to this thread.

It is important that you specify the correct command in this function so that the destination application server executes the correct language handler.

Syntax:

```
CS_RETCODE ASC_send_text(char *server,

char *command,
char *filename,
char *buf)
```

Arguments:

- **server:** The logical name for the application server to where the language request is to be transmitted.
- **command:**Character pointer to a command that is placed in the first line of the transmitted text and is used by the receiving application server to determine the relevant language handler to execute.
- **filename:** The location for the file that is to be transmitted. If you are not transmitting a file, set this to NULL.
- **buf:** Character buffer to be transmitted. If you specify a filename, this buffer is appended to the file contents in the text. If you are not transmitting a buffer, set this to NULL.

Return values:

- **CS_FAIL:** Unable to open a network connection to the destination server. The invocation of dbsqlexec() failed.
- **CS_SUCCEED:** The text was successfully transmitted.

ASC_sendinfo

This function sends an information message to the client.

Syntax:

```
CS_RETCODE ASC_sendinfo(SRV_PROC *sp,

CS_INT msgno,
CS_CHAR *msg)
```

Arguments:

- **sp:** Pointer to the internal thread control structure.
- **msgno:** Message number being sent.
- **msg:** Message text to send.

Return values:

- **CS_SUCCEED:** Message was sent to client.
- **CS_FAIL:** Error occurred sending message.

ASC_set_securedata

Updates or adds user-defined, secure data.

Syntax:

```
CS_RETCODE ASC_set_securedata(char *name,

char*value,
char *desc);
```

Arguments:

- **name:** Key to retrieve the secure data entry.
- **value:** Value (password) of the secure data entry.
- **desc:** Description of the secure data entry.

ASC_spawn

This function sets the server process data segment and then spawns a generic Open Server thread that sets up the application thread.

Using this function, you can pass a message queue name to be created by this function. You can also specify a main function, an exit function, and request that a database connection be established for the spawned thread.

You can access the structure created within the API to manage this spawned thread using the following in-line functions:

- **ASC_SRV_DBPROC** – If you request that a DBPROC be created, this macro points to the DBPROCESS structure allocated by the API for use by this thread.
- **ASC_SRV_MSGQID** – If you request that a message queue be created for this thread, the message queue ID of the message queue created by the API.
- **ASC_SRV_DATA** – If the dataseg_size is not zero, this serves as a pointer to the data segment that is allocated by the API for use by the spawned thread. The thread, or any function called by it, may reference this data segment at any time using this macro.
- **ASC_SRV_STARTUP** – The data segment passed as an argument to the spawned function. This is the start_seg passed to ASC_spawn().

Syntax:

```
CS_RETCODE ASC_spawn(CS_CHAR *name,

int (*main_fn)(VOIDPTR data_seg),
VOIDPTR start_seg,
CS_INT dataseg_size,
CS_CHAR *msgname,
CS_BOOL dbproc_required,
CS_RETCODE (*exit_handler)(SRV_PROC *srvproc))
```

Arguments:

- **name:** Name of the server process.
- **main_fn:** Pointer to the main function of the application thread. When this function terminates, the server process terminates.
- **start_seg:** Pointer to the startup data segment of the server process.
- **dataseg_size:** Size of local data segment required by the application.
- **msgqname:** The message queue that the server process will read. If no message queue is required, set this to NULL.
- **dbproc_required:** Identifies whether or not the server process requires an application database process.
- **exit_handler:** Pointer to the function that is called when the server process terminates. If no server process is required, set this field to NULL.

Return values:

- **CS_SUCCEED:** The spawning of the server process was successful.
- **CS_FAIL:** No server processes were available.

ASC_srv_field_bool

This function calls srv_props to return a field in the server structure:

- **ASC_srv_field_str** – Retrieves strings
- **ASC_srv_field_int** – Retrieves integers
- **ASC_srv_field_bool** – Retrieves boolean

Syntax:

```
CS_BOOL ASC_srv_field_bool(CS_INT property)
```

Arguments:

- **property:** The property to retrieve.

Return values:

- **CS_BOOL:** Boolean value for the server property.

ASC_srv_field_int

This function calls srv_props to retrieve a field in the server structure:

- **ASC_srv_field_str** – Retrieves strings
- **ASC_srv_field_int** – Retrieves integers
- **ASC_srv_field_bool** – retrieves boolean

Syntax:

```
CS_INT ASC_srv_field_int(CS_INT property)
```

Arguments:

- **property:** The property to retrieve.

Return values:

- **CS_INT:** Integer value for the server property.

ASC_srv_field_str

This function calls srv_props to retrieve a field in the server structure:

- **ASC_srv_field_str** – retrieves strings
- **ASC_srv_field_int** – Retrieves integers
- **ASC_srv_field_bool** – Retrieves boolean

Syntax:

```
CS_VOID ASC_srv_field_str(CS_INT property,

CS_CHAR *buf,
CS_INT size)
```

Arguments:

- **property:** The property to retrieve.
- **buf:** Pointer to the destination buffer for string properties.
- **size:** Maximum size of the destination buffer.

ASC_srv_sleep

This is a wrap function around srv_sleep function. ASC adds an extra feature: if sleep is interrupted by a signal of the same variety, the thread continues to sleep.

The thread sleeps until srv_wakeup is called on the same event.

For more information on the srv_sleep function, refer to Sybase documentation.

Syntax:

```
CS_RETCODE ASC_srv_sleep(CS_VOID *sleepeventp,

CS_CHAR *sleeplabelp,
CS_INT sleepflags,
CS_INT *infop,
CS_VOID *reserved1,
CS_VOID *reserved2)
```

Arguments:

- **sleepeventp:** A generic void pointer that srv_wakeup uses to wake the thread or threads. The pointer should be unique for the operating system event that the threads are sleeping on. For example, if a message is passed to another thread, the sending thread could sleep until the message is processed. The pointer to the message would be a useful sleep event that the receiving thread could pass to srv_wakeup to wake the sender.
- **sleeplabelp:** A pointer to a null terminated character string that identifies the event that the thread is sleeping on. This is useful for determining why a thread is sleeping. An application can display this information using the Open Server system registered procedure sp_ps.
- **reserved1:** A platform-dependent handle to a mutex. This argument is ignored on non-preemptive platforms. Set it to (CS_VOID*)0 on non-preemptive platforms.
- **reserved2:** This parameter is not currently used. Set it to 0.
- **sleepflags:** The value of this flag determines the manner in which the thread wakes up.
- **infop:** A pointer to a CS_INT.

For more information on the appropriate values for sleepflags and infop, refer to the Sybase documentation.

ASC_stack_trace

This function prints the stack trace to the server's diagnostic logfile. It calls the Open Server srv_dbg_stack() function to perform this operation.

See also ASC_diag(), ASC_hex_dump(), ASC_rpc_dump().

This function is supported only on platforms where Sybase supports debug capability (this is governed by the SRV_C_DEBUG capability). It is not supported under AIX.

Syntax:

```
void ASC_stack_trace(DIAG_LEVEL level,

char *type,
int line,
char *file)
```

Arguments:

- **level:** The diagnostic level for the function call. See ASC_diag() for more information.
- **type:** Brief description of the circumstances for the function call. It helps to identify such entries in the server's logfile.
- **line:** The line in the source file at which the function was called.
- **file:** The source file from which this function was called.

ASC_thread_field_bool

This function calls srv_thread_props to retrieve a field in the thread structure:

- **ASC_thread_field_str** – Retrieves strings
- **ASC_thread_field_int** – Retrieves integers
- **ASC_thread_field_bool** – Retrieves boolean

Syntax:

```
CS_BOOL ASC_thread_field_bool(SRV_PROC *sp,

CS_INT property)
```

Arguments:

- **sp:** Current thread structure.
- **property:** The property to retrieve.

Return values:

- **CS_BOOL:** Boolean value for the thread property.

ASC_thread_field_int

This function calls srv_thread_props to retrieve a field in the thread structure:

- **ASC_thread_field_str** – Retrieves strings
- **ASC_thread_field_int** – Retrieves integers
- **ASC_thread_field_bool** – Retrieves boolean

Syntax:

```
CS_INT ASC_thread_field_int(SRV_PROC *sp,

CS_INT property)
```

Arguments:

- **sp:** Current thread structure.
- **property:** The property to retrieve.

Return values:

- **CS_INT:** Integer value for the thread property.

ASC_thread_field_str

This function calls srv_thread_props to retrieve a field in the thread structure:

- **ASC_thread_field_str** – Retrieves strings
- **ASC_thread_field_int** – Retrieves integers
- **ASC_thread_field_bool** – Retrieves boolean

Syntax:

```
CS_VOID ASC_thread_field_str(SRV_PROC *sp,

CS_INT property,
CS_CHAR *buf,
CS_INT size)
```

Arguments:

- **sp:** Current thread structure.
- **property:** The property to retrieve.
- **buf:** Pointer to the destination buffer for string properties.
- **size:** Maximum size of the destination buffer.

ASC_threadproc

This function gets the pointer to the server process associated with the current thread.

Syntax:

```
SRV_PROC *ASC_threadproc(void)
```

Return values:

- **SRV_PROC***: Pointer to current server process.

ASC_unlockmutex

This function unlocks mutexes.

See also ASC_lockmutex.

Syntax:

```
CS_RETCODE ASC_lockmutex(SRV_OBJID mutex_id)
```

Arguments:

- **mutex_id:** The mutex ID.

Return values:

- **CS_SUCCEED:** The mutex was unlocked successfully.
- **CS_FAIL:** An error occurred.

ASC_unlock_strtok

Because the UNIX function strtok() maintains a global or static variable of its current position within the string, it is not multi-thread-safe. ASC_lock_strtok is used in conjunction with strtok() in a multithreaded environment. When finished using strtok(), ASC_unlock_strtok() must be called to free the associated mutex.

Syntax:

```
void ASC_unlock_strtok(void)
```

background_process_init

This function initializes the background processes in the server.

Syntax:

```
CS_RETCODE background_process_init (BACKGROUND_PROCESS *tbl)
```

Arguments:

- **tbl:** Table listing the background processes to be started in the server.

Return values:

- **CS_SUCCEED:** The background processes started successfully.
- **CS_FAIL:** An error occurred and the processes could not start.

Server application data types

This section describes all of the data types for the Server Application.

The following list provides an overview of the data types for the Server Application.

- **BACKGROUND_PROCESS:** Defines a background thread or service thread spawned by the application server.
- **LANG_HANDLER:** Describes a language event handler.
- **REG_PROC:** Registered procedure application definition.
- **RPC:** Defines an RPC that will be accepted by the server.
- **RPC_PARAM:** Defines the parameters for a specific RPC.
- **USEREVENT:** Defines user events and the appropriate event handler.

BACKGROUND_PROCESS abstract data type

This structure defines the background or service thread spawned by the application server.

Refer to ASC_spawn() for further details about spawning service threads.

Syntax:

```
typedef struct {

char *qname;
SRV_OBJID *msgqxp;
int (*start)(void *data_seg);
void *data_seg;
BACKGROUND_PROCESS;
```

Members:

- **qname:** Name of the message queue to be created for this service thread.
- **msgqxp:** Pointer to the location where the message queue ID is to be placed.
- **start:** Function pointer to the main thread function.

- **data_seg:** Pointer to a data segment to be made available to the service thread.

LANG_HANDLER abstract data type

This structure describes a language event handler.

Syntax:

```
typedef struct {

char command[LANG_HAND_COMMAND_L+1];
char usage[LANG_HAND_USAGE_L+1];
int (*handler)(SRV_PROC *srvproc, CS_CHAR *langptr,
CS_INT langlen);
LANG_HANDLER;
```

Members:

- **command:** The command associated with the language request. This will be the first line of the language text buffer that is passed to the application server.
- **usage:** The usage of language handler. This usage string is returned to the user in the lang_list RPC.
- **handler:** The function to call to handle this language request. The function syntax is the language handler syntax.

REG_PROC abstract data type

This data type has been superseded by the **CM_RPC** data type.

This data type is a registered procedure application definition.

Syntax:

```
typedef struct {

char *procname;
CS_RETCODE (*handler)(SRV_PROC *srvproc);
REG_PROC_PARAM *param_tbl;
REG_PROC;
```

Members:

- **proname:** Name of the registered procedure being defined.
- **handler:** RPC handler to process the registered procedure when it is executed.
- **param_tbl:** Register procedure parameter table.

RPC abstract data type

Any new code for registered procedures should use the CM_RPC data type instead of this datatype. RPCs continue to use this data type.

This structure defines an RPC that is accepted by the server.

The function add_appl_rpc() is used to add the RPC to the list of RPCs being handled by the server.

Syntax:

```
typedef struct {

CS_CHAR *rpcname;
CS_CHAR *rpcusage;
RPC_PARAM *rpcparams;
CS_RETCODE (*rpchandler)(SRV_PROC *srvproc);
short min_params;
short max_params;
RPC;
```

Members:

- **rpcname:** The name to be used by clients when invoking the RPC (for example, exec SERVER...rpcname).
- **rpcusage:** The usage message that is sent to the user if there is a parameter mismatch.
- **rpcparams:** Parameters necessary to invoke the RPC. Set this to NULL if no parameters are required to invoke the RPC.

  The “rpcparams" field uses the “RPC_PARAM" datatype to specify the parameters used by the RPC.
- **rpchandler:** Pointer to a function to be called when the RPC is received by the library to handle the request. The function will have to accept a pointer to an SRV_PROC structure which contains information describing the server process.
- **min_params:** The minimum number of parameters required by the RPC before it is executed.
- **max_params:** The maximum number of parameters that can be accepted by the RPC before it is executed.

  The “min_params" and “max_params" fields are determined at run-time by the API when it scans the parameter table.

RPC_PARAM abstract data type

This structure defines the parameters for an RPC.

When an RPC is created using the datatype “RPC", you can specify an optional table of RPC parameters. This optional table contains a list of this datatype. Specify the end of the table by setting the **paramname** member to END_PARAM_TABLE.

Any new code should use the CM_RPC_PARAM data type instead of this datatype.

For more information, refer to datatype “RPC".

This structure is used to defined tables and should not be used as a general purpose data structure.

Syntax:

```
typedef struct {

char *paramname;
CS_BOOL null_allowed;
CS_BOOL optional;
int paramtype1;
int paramtype2;
```

```
int paramtype3;
RPC_PARAM;
```

Members:

- **paramname:** Parameter name in the form @name. To mark the end of table, set this parameter to END_PARAM_TABLE.
- **null_allowed:** A boolean field. Specifies whether the parameter allows null values.
- **optional:** A boolean field. Specifies whether or not the parameter can be omitted when the RPC is called. In this case, the processing function for the RPC is generating a default value.
- **paramtype<n>:** These fields specify different parameter types (maximum of three) that are valid for the parameter. The types that can be specified, including CS_CHAR_TYPE, CS_SMALLINT_TYPE, CS_INT_TYPE, etc.

USEREVENT abstract data type

This structure defines user events and the appropriate event handler. To define events, you need to define a table of this structure. To mark the end of the table, set the **event** field to NULL. The function ASC_define_events() is used to define user events to be handled by the server.

Syntax:

```
typedef struct {

int *event;
char *name;
CS_RETCODE (*event_handler)(SRV_PROC *srvproc);
USEREVENT;
```

Members:

- **event:** Pointer to the integer that is used to identify the user-defined event in the server. To mark the end of the table, set this field to **NULL**.
- **name:** The name of the event. This is useful for diagnostics and logs to show the current server event by name.
- **event_handler:** Pointer to the function that is called to handle the event when it occurs in the server.

Client library interface

The client application API enables the Control Server to manage a client application. The client application logic must be written as part of the appl_initialize() function found in the Common Interface API.

A second client library, libclient_external, is also provided. Use libclient_external for client applications that require a separate main() function.

The functions in the client application library are:

- Global variable and termination functions
- Inline functions
- Client application library functions

Global variable

The following global variable is defined by the Client API:

- **SQLSrvName:** The name of the SQL Server.

Termination-related functions

The API functions related to application termination allow application-specific cleanup as part of client application termination.

Inline function

The following inline function is defined by the Client API:

- **SQL Server name:** Terminates the client application.

Client application library functions

The following is a client application library function.

appl_cleanup

This API function allows the client application to perform application-specific cleanup during the termination of the application.

See also appl_initialize() in the Common Interface API.

Syntax:

```
void appl_cleanup(void)
```

Interpreter library

The Interpreter API is used to execute State Tables in ASAP. Within a server, you can define custom State Table actions to enhance the standard State Table language. The functionality described in this section is provided by the API library, libinterpret.

For servers where there are no network elements, you need to define pseudo-network elements with pseudo network element technologies and software generics. For example, when the Interpreter is used in an SRP, the network element host is typically called the name of the SRP, for instance, **CISSRP**, where the technology is **CIS** and the software generic **1.0**.

With the functions described in this chapter you can perform:

- Interpreter initialization
- Action handling

Inline functions

The inline functions and macros specified in this section are used within a State Table action handler. The State Table action handler must be defined with its input argument called **data**.

- **ASDL_CMD:** Identifies the current ASDL command being processed by the Interpreter.
- **CMD_DBG_INFO:** This macro is used to determine whether the debugger is active for the Interpreter. This macro functions as follows: if (CMD_DBG_INFO != NULL) { ... Debugger Active ...}
- **CMD_DIAG:** This macro is used to log diagnostic messages from the State Table action handlers. The macro should be used instead of ASC_diag calls. If the State Table debugger is active, the messages are sent to both the diagnostics file for the server and the trace file for the debugger. If not, the messages are sent to the diagnostics file.
- **CMD_USERID:** Identifies the user who initiated the work order when the Interpreter is used in an NEP.
- **CMD_WO_ID:** Identifies the ASAP work order identifier that the ASDL command belongs to when the interpreter is used in an NEP.
- **CUR_ACT_INT:** Specifies the current action integer to be used by the Interpreter's action handler.
- **CUR_ACT_RECORD:** Identifies the action record being used to process the current State Table action.
- **CUR_ACT_STRING:** Specifies the current action string to be used by the Interpreter's action handler.
- **CUR_ACTION:** Specifies the current State Table action being handled by the Interpreter.
- **CUR_LINE:** Specifies the current State Table program line number being executed by the Interpreter.
- **GEN_RESPONSE_FILE ( ):** Generates the UNIX filename to be used to store the switch response.
- **HOST_CLLI:** Identifies the network element host associated with the Interpreter.
- **PROGRAM_COUNTER:** Specifies the current program counter of the Interpreter. The State Table action handler changes this value before returning the control to the Interpreter.
- **RESPONSE_FILE:** Identifies the UNIX file that is used by the command processor for storing the switch response to a network element command. This is only meaningful when used in an NEP.
- **SFTWR_LOAD:** Identifies the software version of the network element.
- **SRQ_ID:** Specifies the service request identifier associated with the ASDL when the Interpreter is used in a NEP.
- **TECH:** Identifies the technology of the network element (for instance, DMS).

Interpreter library functions

This section describes the functions in the Interpreter Library.

ASC_alloc_Interpreter

This function allocates and initializes an Interpreter data segment and sets it up to be used by an application that is not the command processor.

Syntax:

```
CMD_PROC_DATA *ASC_alloc_Interpreter(CS_CHAR *host_clli,

CMD_PROC_MSG *msg,
PORT_BIND_ST *port)
```

Arguments:

- **host_clli:** Determines the technology and software version.
- **msg:** Pointer to the command processor message. Can be null.
- **port:** Pointer to the Port Bind structure. Can be null.

Return values:

- **CMD_PROC_DATA***: Pointer to the Interpreter data segment.
- **NULL:** An error occurred.

ASC_delete_int_var

This function deletes the Interpreter variable you specify.

This is an inline function.

Syntax:

```
CS_RETCODE ASC_delete_int_var(CMD_PROC_DATA *data,

char *label)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **label:** Interpreter variable label.

ASC_free_Interpreter

This function releases the memory associated with the Interpreter data segment.

Syntax:

```
void ASC_free_Interpreter(CMD_PROC_DATA *data)
```

Arguments:

- **data:** Interpreter data segment.

ASC_get_dev_sess_data

This function gets the device session information structure from the Interpreter.

This is an inline function.

See also ASC_set_dev_sess_data.

Syntax:

```
VOIDPTR ASC_get_dev_sess_data(COMM_DATA_ST *comm_data)
```

Arguments:

- **comm_data:** Pointer to the communication data structure within the Interpreter data segment.

Return values:

- **dev_sess_data:** Pointer to the device-specific information data structure.

ASC_get_int_appl_data

This function retrieves the application data segment for the Interpreter.

This is an inline function.

Syntax:

```
VOIDPTR ASC_get_int_appl_data(CMD_PROC_DATA *data)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.

Return values:

- **VOIDPTR:** Application data segment in use for the Interpreter.

ASC_get_int_var

This function retrieves the value for the Interpreter variable you specify.

This is an inline function.

Syntax:

```
CS_RETCODE ASC_get_int_var(CMD_PROC_DATA *data,

char *label,
char *value)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **label:** Interpreter variable label.
- **value:** Variable to save the value in.

Return values:

- **CS_SUCCEED:** Interpreter variable retrieval was successful.
- **CS_FAIL:** The retrieval failed.

ASC_init_Interpreter

This function initializes the Interpreter subsystem within a server. This initialization includes starting the cache manager, initializing the regular expression system, initializing the database process pool, etc.

The Interpreter can only be initialized once per process. Subsequent calls to ASC_init_Interpreter() are ignored.

Syntax:

```
void ASC_init_Interpreter(CS_CHAR *debug_host,

DEBUG_CONFIGURATION_ST *dbg_cfg)
```

Arguments:

- **debug_host:** String that specifies the host CLLI that is used for debugging. It is currently not required.
- **dbg_cfg:** Field that specifies the debug configuration structure for the server. For servers that use an Interpreter other than an NEP, it is set to NULL.

ASC_Interpreter

This function sets up the appropriate data fields in the Interpreter data segment and calls the main Interpreter function to execute the State Table.

Syntax:

```
CS_RETCODE ASC_Interpreter(CMD_PROC_DATA *data,

CS_CHAR *asdl_cmd,
CS_BOOL auto_free)
```

Arguments:

- **data:** Command processor data segment.
- **asdl_cmd:** ASDL command to be executed. The ASDL command is directly related to technology and software load, and this determines the State Table (as defined in tbl_nep_asdl_prog).
- **auto_free:** Boolean flag. Indicates if the Interpreter parameters are automatically made available upon completing the ASDL command.

Return values:

- **CS_SUCCEED:** State Table execution for the ASDL command was successful.
- **CS_FAIL:** State Table execution failed for the ASDL command.

ASC_set_dev_sess_data

This function sets the device session information structure for the Interpreter.

This is an inline function.

See also ASC_get_dev_sess_data.

Syntax:

```
void ASC_set_dev_sess_data (CMD_PROC_DATA *data,

VOIDPTR *dev_sess_data)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **dev_sess_data:** Void pointer to a device-specific information structure.

ASC_set_int_appl_data

This function sets the application data segment for the Interpreter.

This is an inline function.

Syntax:

```
void ASC_set_int_appl_data(CMD_PROC_DATA *data,

VOIDPTR *appl_seg)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **appl_seg:** Pointer to a user-defined application data segment.

ASC_store_int_var

This function saves the Interpreter variable in the Interpreter application data segment.

Syntax:

```
CS_RETCODE ASC_store_int_var(CMD_PROC_DATA *data,

char *label,
char *value)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **label:** Interpreter variable label.
- **value:** Variable to save the value in.

Return values:

- **CS_SUCCEED:** The variable was saved in the Interpreter application data segment successfully.
- **CS_FAIL:** The save failed.

CMD_delete_var

This function deletes a variable in the Interpreter data segment.

Syntax:

```
CS_RETCODE CMD_delete_var(CMD_PROC_DATA *data,
```

```
char *label)
```

Arguments:

- **data:** Pointer to the Interpreter data segment.
- **label:** Interpreter variable label.

Return values:

- **CS_SUCCEED:** The variable was deleted successfully in the Interpreter data segment.
- **CS_FAIL:** The deletion failed.

CMD_expand_action_string

This function copies the current action string into the buffer you specify and performs variable substitution.

Syntax:

```
CS_RETCODE CMD_expand_action_string(CMD_PROC_DATA *data,

char *buf)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.
- **buf:** Pointer to the buffer that holds the expanded action string.

Return values:

- **CS_SUCCEED:** Successfully expanded the action string to the specified buffer.
- **CS_FAIL:** Could not expand the action string.

CMD_free_assignment

This function releases the memory for a specified assignment buffer that has been allocated using the CMD_get_assignment function.

Syntax:

```
void CMD_free_assignment(CMD_ASSIGNMENT_BUF *buf)
```

Arguments:

- **buf:** Pointer to the assignment buffer to be freed.

CMD_free_bvar_assignment

This function releases the memory for an assignment buffer that has been allocated using the CMD_get_bvar_assignment function.

See also CMD_get_bvar_assignment.

Syntax:

```
void CMD_free_bvar_assignment(CMD_BVAR_ASSIGNMENT_BUF *buf)
```

Arguments:

- **buf:** Pointer to the assignment buffer to be freed.

CMD_free_dbproc

This function frees the database process.

Syntax:

```
void CMD_free_dbproc(DBPROCESS *dbproc)
```

Arguments:

- **dbproc:** Pointer to the DBPROCESS.

CMD_get_assignment

This function allocates an assignment buffer, parses the current action string, and stores the results in the buffer of a linked list. The primary difference between this function and the CMD_parse_assignment is that, with this function, the number of arguments is unlimited. If an error is detected, NULL is returned.

Variable substitution is performed when you specify variables in the action string. The format of the action string is as follows:

```
<arg>::=constant/%var
%var::=<arg1>:<arg2>:..:<argN>
```

It is the caller's responsibility to free the allocated buffer (which is a linked list) by calling CMD_free_assignment after processing is done.

Syntax:

```
CMD_ASSIGNMENT_BUF *CMD_get_assignment (CMD_PROC_DATA *data)
```

Arguments:

- **data:** Pointer to Interpreter data segment.

Return values:

- **Pointer to assignment buffer:** Pointer to the retrieved assignment buffer.

CMD_get_bvar

This function scans the variable table for the variable you specify and then returns the appropriate value and status.

Syntax:

```
CS_RETCODE CMD_get_bvar(CMD_PROC_DATA *data,

CS_CHAR *label,
CS_VOID **value,
CS_INT *len,
CS_VOID **template)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.

- **label:** Field that specifies the name of the binary variable.
- **value:** Indirect pointer to the data buffer in which the binary value is saved (return parameter).
- **len:** Integer pointer to save length of the buffer (return parameter).
- **template:** Indirect pointer to the binary data template structure. If used, it describes the fields and structure of the binary variable (return parameter).

Return values:

- **CS_SUCCEED:** The variable is defined and its value has been stored in the destination specified.
- **CS_FAIL:** The variable does not exist or it is not a binary variable.

CMD_get_bvar_assignment

This function allocates an assignment buffer, parses the current action string, and stores the results in the buffer of a linked list. A difference between this function and the CMD_get_assignment is that, with this function, the linked list node can hold binary data.

Variable substitution is performed when you specify variables in the action string. The format of the action string is as follows:

```
<arg>::=constant/ASCII variable/Binary variable
%var::=<arg1>:<arg2>:..:<argN>
```

It is the caller's responsibility to free the allocated buffer (which is a linked list) by calling CMD_free_bvar_assignment after processing is done.

Syntax:

```
CMD_BVAR_ASSIGNMENT_BUF *CMD_get_bvar_assignment(CMD_PROC_DATA *data)
```

Arguments:

- **data:** Pointer to Interpreter data segment.

Return values:

- **Assignment buffer:** Pointer to assignment buffer.

CMD_get_var

This function scans the variable table for the ASCII variable you specify and then returns the value and status.

See also "CMD_store_var ."

Syntax:

```
CS_RETCODE CMD_get_var(CMD_PROC_DATA *data,

char *label,
char *value)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.

- **label:** Name of the variable.
- **value:** Buffer where the value of the variable is to be stored.

Return values:

- **CS_SUCCEED:** The variable is defined and its value has been stored in the destination specified.
- **CS_FAIL:** The variable does not exist or is not an ASCII variable.

CMD_lock_regexpr

This function locks the mutex associated with regular expression management.

See also "CMD_unlock_regexpr."

Syntax:

```
void CMD_lock_regexpr(void)
```

CMD_parse_assignment

This function is superseded by CMD_get_assignment() which has no limit on the number of arguments.

This function scans the current action string to locate the destination variable and the three arguments for the action function. The format for the action string is as follows:

```
%var ::=<arg1>:<arg2>:<arg3>
where <arg> is a variable <%var> or a value (for example, %x=%y:10:%z)
```

Syntax:

```
CS_RETCODE CMD_parse_assignment(CMD_PROC_DATA *data,

PARSE_BUF *buf)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.
- **parse_buf:** Pointer to a parse buffer.

Return values:

- **CS_SUCCEED:** Parse of the string was successful.
- **CS_FAIL:** Parse of the string failed.

CMD_store_bvar

This function stores information about a binary ASDL program variable in the command processor or program variable table or updates the variable if it already exists in the table.

See also "CMD_get_bvar."

Syntax:

```
CS_RETCODE CMD_store_bvar(CMD_PROC_DATA *data,
```

```
CS_CHAR *label,
CS_VOID *value,
CS_INT len,
CS_VOID * void (*template_destructor)(CS_VOID *template))
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.
- **label:** Name of the binary variable.
- **value:** Void pointer to the data buffer in which the binary value is saved.
- **len:** Length of the buffer.
- **template:** Pointer to the binary data template structure. If used, it describes the fields and structure of the binary variable.
- **template_destructor:** Pointer to the destructor function to be called to deallocate the template. If it is not required, set it to NULL.

Return values:

- **CS_SUCCEED:** The variable was successfully stored or updated in the variable table.
- **CS_FAIL:** The variable was not stored in the variable table.

CMD_store_var

This function stores an ASCII ASDL program variable in the command processor or program variable table. If the program variable already exists in the table, this function updates it.

See also "CMD_get_var."

Syntax:

```
CS_RETCODE CMD_store_var(CMD_PROC_DATA *data,

char *label,
char *value)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.
- **label:** Name of the variable.
- **value:** Value of the variable.

Return values:

- **CS_SUCCEED:** The variable was successfully stored or updated in the variable table.
- **CS_FAIL:** The variable was not stored in the variable table.

CMD_store_zero_pad_var

This function stores an ASCII ASDL program variable in the command processor or program variable table. If the program variable already exists in the table, this function updates it. In addition, this function formats the variable based on the maximum field

length, padding the field with leading zeroes. If the string length is greater than the total field length specified, the truncated value is saved in the variable.

See also "CMD_get_var."

Syntax:

```
CS_RETCODE CMD_store_zero_pad_var(CMD_PROC_DATA *data,

char *label,
char *value,
CS_INT total_field_len zero_pad)
```

Arguments:

- **data:** Pointer to the local data segment for the command processor.
- **label:** Name of the variable.
- **value:** Value of the variable.
- **zero_pad:** Maximum field length to be used when formatting this numerical field.

Return values:

- **CS_SUCCEED:** The variable was successfully stored or updated in the variable table.
- **CS_FAIL:** The variable was not stored in the variable table.

CMD_unlock_regexpr

This function unlocks the mutex associated with regular expression management.

See also "CMD_lock_regexpr."

Syntax:

```
void CMD_unlock_regexpr(void)
```

CMD_user_actions

This function adds an action to the action table in the main Interpreter State Table action tree. If the action already exists, the new action overrides the existing action.

Syntax:

```
CS_RETCODE CMD_user_actions(ACTION_RECORD *action_tbl)
```

Arguments:

- **action_tbl:** User action table. Must be terminated with the last entry having an action equal to NULL.

Return values:

- **CS_SUCCEED:** User-specific State Table action successfully added.
- **CS_FAIL:** User actions could not be installed into the Interpreter action table.

Control configuration interface

This section describes the functions for the Control subsystem.

The Control subsystem supports static table configuration. Use functions instead of SQL insert scripts to interface with the static configuration database tables.

The function-based interface reduces the dependency between administrators who configure the system and product developers who need to make changes to the static tables to support new functionality.

> **Note:**
>
> If you invoke an CSP_del_* function without parameters, all rows in the table are deleted.
>
> If you invoke the CSP_list_* functions without parameters, all rows are listed.

Interface definitions

This section lists the syntax, descriptions, parameters, and results for Control configuration actions.

CSP_db_admin

This function purges all performance data that have been stored for more than a specified number of days. The default value of a_days is 3 days if it is not provided.

For more information about using functions, see "Oracle Execution Examples."

Affected tables:

- tbl_alarm_log
- tbl_event_log
- tbl_process_info

Table 3-2 CSP_db_admin Parameters

Name	Description	Req'd	(I)nput/ (O)utput
days	Specifies the age (in days) of log data to delete. All data older than the specified number of days is deleted.	Yes	I

CSP_del_alarm

This function deletes a system alarm code from tbl_system_alarm.

For more information about using functions, see "Oracle Execution Examples."

Table 3-3 CSP_del_alarm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
alarm_code	System alarm code identifier.	No	I

CSP_del_appl

This function deletes ASAP application registration information from the Control database (tbl_appl_proc).

For more information about using functions, see "Oracle Execution Examples."

Table 3-4 CSP_del_appl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
appl_cd	Logical name of the ASAP application server.	No	I

CSP_del_center

This function deletes an alarm center definition from the control database (tbl_alarm_center).

For more information about using functions, see "Oracle Execution Examples."

Table 3-5 CSP_del_center Parameters

Name	Description	Req'd	(I)nput/ (O)utput
alarm_center	The alarm center to be deleted.	No	I

CSP_del_code

This function deletes an administration system code from the database (tbl_code_list).

For more information about using functions, see "Oracle Execution Examples."

Table 3-6 CSP_del_code Parameters

Name	Description	Req'd	(I)nput/ (O)utput
code_type	Type of code. For example: “DB": database script related entry.	No	I
code	The code.	No	I
value	Value of the code.	No	I

CSP_del_component

This function deletes an ASAP component from tbl_component.

For more information about using functions, see "Oracle Execution Examples."

Table 3-7 CSP_del_component Parameters

Name	Description	Req'd	(I)nput/ (O)utput
territory	The ASAP territory.	No	I
system	The ASAP system.	No	I
component	The ASAP component.	No	I

CSP_del_db_thresh

This function deletes a database threshold definition from tbl_db_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 3-8 CSP_del_db_thresh Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asap_sys	The ASAP environment (“TEST", “PROD", etc.).	No	I
db_name	The database name.	No	I

CSP_del_event

This function deletes an ASAP event type from the database (tbl_event_type).

For more information about using functions, see "Oracle Execution Examples."

Table 3-9 CSP_del_event Parameters

Name	Description	Req'd	(I)nput/ (O)utput
event_type	The event type. For example, “ABNORMAL", “SYS_ERR", etc.	No	I

CSP_del_fs_thresh

This function deletes a file system threshold definition from tbl_fs_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 3-10 CSP_del_fs_thresh Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asap_sys	The ASAP environment (TEST, PROD, etc.).	No	I
file_system	The UNIX file system for which the threshold definition is to be deleted.	No	I

CSP_del_listener

This function deletes a listener entry from tbl_listeners.

For more information about using functions, see "Oracle Execution Examples."

Table 3-11 CSP_del_listeners Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srv_name	Name of the server that starts a socket listener. The SARM must start a socket listener to receive incoming Java SRP requests. For a Java-enabled NEP, this is the name of the NEP ($NEP). For the Java SRP, this column contains the SARM name.	Yes	I
listener_name	The name of the listener thread. For a Java-enabled NEP, the listener name describes the listener in the Java process that accepts interpreter requests from the C process. This listener name must always be $NEP_jlistener. For the Java SRP, observe the naming convention of “<Java SRP application name>_jsrplistener". This column is used by the Java SRP to retrieve the listener configurations.	Yes	I

CSP_del_nvp

This function deletes a name/value pair from the database (tbl_name_value_pair).

For more information about using functions, see "Oracle Execution Examples."

Table 3-12 CSP_del_nvp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
name	Name of the name/value pair.	No	I

CSP_get_listener

This function lists listener entries associated with an NEP (tbl_listeners).

For more information about using functions, see "Oracle Execution Examples."

Table 3-13 CSP_get_listener Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srv_name	Name of the server that starts a socket listener. The SARM must start a socket listener to receive incoming Java SRP requests. For a Java-enabled NEP, this is the name of the NEP ($NEP). For the Java SRP, this column contains the SARM name.	Yes	I

CSP_list_alarm

This function lists system alarms contained in tbl_system_alarm.

For more information about using functions, see "Oracle Execution Examples."

Table 3-14 CSP_list_alarm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
alarm_code	System alarm code.	No	I

Table 3-15 CSP_list_alarm Results

Name	Datatype	Description
alarm_code	TYP_code	The alarm code.
description	TYP_desc	Brief description of the system alarm.
alarm_level	TYP_alarm_level	Level of the alarm.
escalation_code	TYP_code	Escalation code of the alarm.
escalation_time	TYP_time	Escalation time.
auto_clear	TYP_yes_no	Determines whether the alarm must be automatically cleared.
route#_period	TYP_short	Interval in minutes for the alarm to be sent to the alarm center. # can be a number between 1 and 5 to designate up to five routings.
route#_start	TYP_time	Daily start time in minutes after midnight. # can be a number between 1 and 5 to designate up to five routings.
route#_end	TYP_time	Daily end time in minutes after midnight. # can be a number between 1 and 5 to designate up to five routings.

Table 3-15 (Cont.) CSP_list_alarm Results

Name	Datatype	Description
route#_center	TYP_code	Alarm center to route alarm to. # can be a number between 1 and 5 to designate up to five routings.

CSP_list_appl

This function lists ASAP application registration information for the specified appl_cd or all applications from the Control database (tbl_appl_proc).

For more information about using functions, see "Oracle Execution Examples."

Table 3-16 CSP_list_appl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
appl_cd	The logical name of the ASAP application server.	No	I

Table 3-17 CSP_list_appl Results

Name	Datatype	Description
start_seq	TYP_start_seq	Controls the sequence in which the applications are started. For example, certain client applications may be required to start before server applications, and other client applications after the server applications.
appl_type	TYP_appl_type	Specifies whether the ASAP application is an application server or a client application. Specify: • S – For server • C – For client • M – For master control server • R – For remote slave control server
appl_cd	TYP_code	ASAP logical client/server name, for example, SARM, NEP01, NEP02.
control_svr	TYP_code	The logical ASAP application control server that spawns this application and monitors its behavior.
description	TYP_desc	Brief description of the ASAP application.
diag_file	TYP_unix_file	The name of the diagnostics logfile to which diagnostic messages are written. This file is created in the $LOGDIR directory under a dated directory, for example, in the $LOGDIR/yymmdd format.
auto_start	TYP_yesno	An autostart flag that determines if the application is to be started automatically when ASAP starts.
program	varchar(40)	The name of the UNIX program that executes to start the ASAP application. The UNIX program must reside in the $PROGRAMS directory and be executable.

Table 3-17 (Cont.) CSP_list_appl Results

Name	Datatype	Description
diag_level	TYP_diag_level	The diagnostic level of the ASAP application. The diagnostic level is used to determine whether or not to log diagnostic information based on the diagnostic level of the ASC_diag() API function call.
isactive	TYP_yesno	A yes/no flag denoting whether the ASAP server is currently active.
last_start	datetime	The last start date and time of the ASAP server.
last_halt	datetime	The last halt or terminate date and time of the ASAP server.
last_abnormal	datetime	The last abnormal termination of the Control server.
svr_type	varchar(8)	This field defines the type of server.

CSP_list_center

This function lists alarm center definitions from the control database (tbl_alarm_center).

For more information about using functions, see "Oracle Execution Examples."

Table 3-18 CSP_list_center Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
alarm_center	The alarm center to be deleted.	No	I

Table 3-19 CSP_list_center Results

Name	Datatype	Description
alarm_center	TYP_code	The unique code representing the alarm center.
control_prog	TYP_unix_file	The program to be executed to communicate the alarm to the alarm center.
description	TYP_desc	Brief description of the alarm center.
opt#_type	TYP_option	First option to the control program, where # represents a value between 1 and 5.
opt#_value	TYP_opt_value	Argument to the first option, where # represents a value between 1 and 5.

CSP_list_code

This function lists Administration System code(s) from tbl_code_list.

If you invoke the function without any parameters, all rows in the table are listed.

For more information about using functions, see "Oracle Execution Examples."

Table 3-20 CSP_list_code Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
code_type	Type of code. For example, DB for a database script related entry.	No	I
code	The code entry.	No	I
value	Value of the code.	No	I

Table 3-21 CSP_list_code Results

Name	Datatype	Description
code_type	TYP_code_type	Type of code.
code	TYP_code_text	Type of code.
value	TYP_code_value	Parameter value associated with the label.
code_desc	TYP_desc	Brief description of the code.
parm#	TYP_code_parm	General purpose parameter, where # represents a value between 1 and 4.

CSP_list_component

This function lists ASAP components contained in tbl_component.

If you invoke the function without any parameters, all rows in the table are listed.

For more information about using functions, see "Oracle Execution Examples."

Table 3-22 CSP_list_component Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
territory	Identifies the ASAP territory.	No	I
system	Identifies the ASAP system.	No	I
component	Identifies the ASAP component.	No	I

Table 3-23 CSP_list_component Results

Name	Datatype	Description
territory	varchar(20)	ASAP territory.
system	varchar(20)	ASAP system.
component	varchar(40)	ASAP component.

CSP_list_db_thresh

This function lists database threshold definition(s) contained in tbl_db_threshold.

If you invoke the function without any parameters, all rows in the table are listed.

For more information about using functions, see "Oracle Execution Examples."

Table 3-24 CSP_list_db_thresh Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asap_sys	The ASAP environment (for example, TEST, PROD etc.)	No	I
db_name	Database name.	No	I

Table 3-25 CSP_list_db_thresh Results

Name	Datatype	Description
asap_sys	TYP_code	ASAP environment.
db_name	varchar(80) TYP_desc	Database name.
data_threshold	int	Database threshold, in Mb.
tran_threshold	int	Transaction log threshold.
data_event	TYP_code	Event issued if the database threshold is exceeded.
tran_event	TYP_code	Event to be issued if the transaction log threshold is exceeded.

CSP_list_event

This function lists ASAP event definitions contained in tbl_event_type.

If you invoke the function without any parameters, all rows in the table are listed.

For more information about using functions, see "Oracle Execution Examples."

Table 3-26 CSP_list_event Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
event_type	Event type. For example, the core uses some of the following event types: ABNORMAL, SYS_ERR, etc.	No	I

Table 3-27 CSP_list_event Results

Name	Datatype	Description
event_type	TYP_event	The event type. The core module includes some of the following event types: ABNORMAL, SYS_ERR, etc.
description	varchar(40) TYP_desc	Brief description of the event.
alarm_code	TYP_code	The alarm code associated with the event.
alarm_action	TYP_alarm_action	Specifies whether the alarm is enabled or disabled.

CSP_list_fs_thresh

This function lists file system threshold definitions contained in tbl_fs_threshold.

If you invoke the function without any parameters, all rows in the table are listed.

For more information about using functions, see "Oracle Execution Examples."

Table 3-28 CSP_list_fs_thresh Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asap_sys	The ASAP environment (TEST, PROD, etc.).	No	I
file_system	The UNIX file system for which the threshold definition is to be deleted.	No	I

Table 3-29 CSP_list_fs_thresh Results

Name	Datatype	Description
asap_sys	TYP_code	The ASAP environment.
file_system	varchar(100) TYP_desc	File system name.
full_threshold	int	File system full threshold.
full_event	TYP_code	Event to be generated if the full threshold is exceeded.

CSP_list_nvp

This function lists name/value pairs from the database (tbl_name_value_pair).

If you invoke the function without any parameters, all rows in the table are deleted.

For more information about using functions, see "Oracle Execution Examples."

Table 3-30 CSP_list_nvp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
name	Name of the name-value pair.	No	I

Table 3-31 CSP_list_nvp Results

Name	Datatype	Description
name	varchar(40)	Name of the parameter.
value	int	Parameter value associated with the label.

CSP_new_alarm

This function defines a system alarm which may be generated by ASAP system events. This includes the start time, interval, and end time for the alarm.

This function populates tbl_system_alarm.

Example:

The following example creates an alarm for the abnormal termination of an application process to the ADMINPGR center that is continuous on a five minute period any time of the day, type the following:

```
var retval number;
exec :retval := CSP_new_alarm ('ABNORMAL', 'Abnormal process termination',
'CRITICAL', '',NULL, 'N', 5, 0, 1440, 'ADMINPGR');

print retval;
```

For more information about using functions, see "Oracle Execution Examples."

Table 3-32 CSP_new_alarm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
alarm_code	System alarm code.	Yes	I
description	Brief description of the system alarm.	Yes	I
alarm_level	The level of the alarm, for example, MAJOR, MINOR, CRITICAL.	Yes	I
escalation_code	The escalation code for the alarm if the alarm is not corrected within the escalation time (defined next).	Yes	I
escalation_time	Time after the alarm was raised that the escalation of the alarm must take place.	Yes	I
auto_clear	Flag that determines if the alarm should be automatically cleared upon generation.	Yes	I

Table 3-32 (Cont.) CSP_new_alarm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
route#_period	Interval in minutes for the alarm to be sent to the alarm center, where # represents a value between 1 and 5.	Yes (if # = 1), No (if # = 2 to 5)	I
route#_start	The daily start time in minutes since midnight for alarms to go to this alarm center, where # represents a value between 1 and 5.	Yes (if # = 1), No (if # = 2 to 5)	I
route#_end	The daily end time in minutes since midnight, where # represents a value between 1 and 5.	Yes (if # = 1), No (if # = 2 to 5)	I
route#_center	Alarm center to route alarm to, where # represents a value between 1 and 5.	Yes (if # = 1), No (if # = 2 to 5)	I

CSP_new_appl

This function defines a new ASAP client or server application in tbl_appl_proc.

For more information about using functions, see "Oracle Execution Examples."

Table 3-33 CSP_new_appl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
start_seq	Specifies the ASAP startup sequence. This determines the sequence in which applications are started.	Yes	I
appl_type	Specifies the ASAP application server: • S – For server • C – For client • M – For master control server • R – For remote slave control server	Yes	I
appl_cd	The logical ASAP application code, for example, SARM, NEP01, NEP02.	Yes	I
control_svr	Logical ASAP Control server.	Yes	I
auto_start	An autostart flag.	Yes	I
program	The name of the UNIX program to execute to start the ASAP application.	Yes	I
diag_level	The diagnostic level of the ASAP application. This is used to determine whether or not to log certain diagnostic information based on the diagnostic level of the **ASC_diag()** API function call. See the associated rule for possible values.	Yes	I

Table 3-33 (Cont.) CSP_new_appl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
diag_file	The name of the diagnostics file in which diagnostic messages are to be placed. This file is created in the $LOGDIR directory under a dated directory, for example, using the $LOGDIR/yymmdd format.	Yes	I
description	A description of the ASAP application.	Yes	I
svr_type	ASAP server type. Possible values include: • CTRL – Control server • MASTER – Master Control server (must be only one per system) • SARM – SARM server • SRP – SRP server • NEP – NEP server • OTHER	No	I

Example:

To configure ASAP to manage and monitor this processes, enter the following commands. This example assumes this processes is executed automatically at startup and the **Low** level diagnostics are active.

```
var retval number;
exec :retval := CSP_new_appl (4, 'S', 'NEPAXE', 'CONTROL2','Y', 'LOW',
'NEPAXE.diag', 'NEP for AXE Switches')
```

Once the control servers are started on their respective machines, you can start and shut down the application processes automatically from the master system.

CSP_new_center

This function defines an alarm center to which alarm notifications are sent by the Control server. This function populates tbl_alarm_center.

Syntax:

```
var retval number;
exec :retval :=  CSP_new_center ('alarm_center', 'control_prog',
['description'], ...);
```

Example:

In the following example, **admin.sh** is the ADMIN center and **adminpg** is the ADMINPGR center. You must place the final versions of these programs in the ASAP programs directory and identify them using the environment variable **$PROGRAMS**.

```
var retval number;
exec :retval :=  CSP_new_center ('ADMIN', 'admin.sh', 'General Admin. Center');
```

For more information about using functions, see "Oracle Execution Examples."

Table 3-34 CSP_new_center Parameters

Name	Description	Req'd	(I)nput/ (O)utput
alarm_center	A unique code representing the alarm center.	Yes	I
control_prog	The program to be executed to communicate the alarm to the alarm center.	Yes	I
description	A brief description of the alarm center or control program.	No	I
opt#_type	Option name to be passed to the control program, where # represents a value between 1 and 5.	No	I
opt#_value	Value of the option, where # represents a value between 1 and 5.	No	I

CSP_new_code

This function populates tbl_code_list with core or custom code used by ASAP. For instance, this function to identify codes that track the cartridges deployed within ASAP.

For more information about using functions, see "Oracle Execution Examples."

Table 3-35 CSP_new_code Parameters

Name	Description	Req'd	(I)nput/ (O)utput
code_type	Type of code. For example, DB for database script related entry.	Yes	I
code	The code.	Yes	I
value	Value of the code.	Yes	I
code_desc	Brief description of the code.	No	I
parm#	General purpose parameter, where # can be a number between 1 and 4.	No	I

CSP_new_component

This function defines an ASAP component in a territory and adds it to database table tbl_component.

For more information about using functions, see "Oracle Execution Examples."

Table 3-36 CSP_new_component Parameters

Name	Description	Req'd	(I)nput/ (O)utput
territory	Identifies the ASAP territory.	Yes	I
system	Identifies the ASAP system.	Yes	I
component	Identifies an ASAP system component within a territory and system. For example, SRP, SARM, etc.	Yes	I

CSP_new_db_thresh

This function defines database and/or transaction log thresholds to be used by the Control server, and writes the information to tbl_db_threshold. The Control server monitors the database/transaction log size and issues the appropriate data event/ transaction event when the threshold is exceeded.

Syntax:

```
var retval number;
exec :retval :=  CSP_new_db_thresh ('asap_sys', 'db_name', 'db_threshold',
'tran_threshold', 'data_event', 'tran_event')
```

Example:

```
var retval number;
exec :retval :=  CSP_new_db_thresh ('PROD', 'SDB_P01_asap', 80, 20, 'DB2FULL',
'TRANFULL';
```

in this example, the database threshold is set to 80 percent. If the database becomes more than 80 percent full, the DB2FULL system event is issued. This command also sets the transaction log threshold for the database. When the transaction log for the database exceeds 20 MB, the TRANFULL system event is issued.

Database thresholds must be defined in the component table.

For more information about using functions, see "Oracle Execution Examples."

Table 3-37 CSP_new_db_thresh Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asap_sys	The ASAP environment (for example, TEST, PROD, etc.).	Yes	I
db_name	Database name.	Yes	I
db_threshold	Database threshold, in Mb.	Yes	I
tran_threshold	Transaction log threshold.	Yes	I
data_event	Data event.	Yes	I
tran_event	Transaction log event.	Yes	I

CSP_new_event

This function defines an “event type" within ASAP, and optionally, the associated “alarm code" that is associated with the event. System events can be generated when an error condition is encountered in ASAP. CSP_new_event populates tbl_event_type.

For more information about using functions, see "Oracle Execution Examples."

Table 3-38 CSP_new_event Parameters

Name	Description	Req'd	(I)nput/ (O)utput
event_type	Event type. For example, the core uses some of the following event types: ABNORMAL, SYS_ERR, etc.	Yes	I
description	A brief description of the ASDL command.	No	I
alarm_code	The system alarm code associated with the event. If NULL, no alarm will be generated and only the database log entry will be created.	No	I
alarm_action	Specifies the alarm action, and may specify whether the alarm is presently enabled or disabled.	No	I
notify_aims	This is not currently implemented.		

Syntax:

```
var retval number;
exec :retval :=  CSP_new_event ('event_type', ['description'], ['alarm_code'],
["alarm_action'];
```

Example:

The following example shows an ABNORMAL system event mapped to the ABNORMAL alarm.

```
var retval number;
exec :retval :=  CSP_new_event ('ABNORMAL', 'Abnormal Process Termination Event',
'ABNORMAL', 'E';
```

CSP_new_fs_thresh

This function defines a file system threshold to be used by the Control server. The Control server monitors the file system size and if the threshold is exceeded, the appropriate system event is generated. File system threshold information is stored in tbl_fs_threshold.

For more information about using functions, see "Oracle Execution Examples."

CSP_new_listener

This function adds a listener entry to tbl_listeners. You must configure this table to allow the SARM to start up socket listeners for incoming SRP requests. As well, every Java-enabled NEP must maintain a dedicated connection to its JInterpreter.

For more information about using functions, see "Oracle Execution Examples."

Table 3-39 CSP_new_listener Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srv_name	Name of the server that starts a socket listener. The SARM must start a socket listener to receive incoming Java SRP requests. For a Java-enabled NEP, this is the name of the NEP ($NEP). For the Java SRP, this column contains the SARM name.	Yes	I
host_name	The host name or the IP address on which the server application resides. For the JInterpreter, this value must always be localhost. For the Java SRP, the host_name identifies the location of the SARM.	Yes	I
listener_name	The name of the listener thread. For a Java-enabled NEP, the listener name describes the listener in the Java process that accepts interpreter requests from the C process. This listener name must always be $NEP_jlistener. For the Java SRP, observe the naming convention of “<Java SRP application name>_jsrplistener". This column is used by the Java SRP to retrieve the listener configurations.	Yes	I
port	A free port on which the server can start the socket listener.	Yes	I

CSP_new_nvp

This function defines parameters (name value pairs) that are required to maintain the control database. Typical parameters include “audit trail window", “log retention window", etc. This information is stored in tbl_name_value_pair.

For more information about using functions, see "Oracle Execution Examples."

Table 3-40 CSP_new_nvp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
name	Name of the parameter.	Yes	I
value	Value of the parameter.	Yes	I

Object oriented (OO) common library

This section describes the design of an object-oriented programming interface (class library) that forms the general framework for building object-oriented multithread-safe applications.

An application framework manages initialization and cleanup when you start up and shut down the system. In addition, it provides basic facilities, including managing connections to servers, handling client and server messages, retrieving application configurations, logging diagnostic messages to appropriate files, and generating system events on server side, etc., for use within the application.

This object-oriented interface is designed to fulfill the above requirements with guaranteed multithread safety on these operations in a multithreaded environment. Using the interface in the object-oriented applications hides the detailed operations behind an interface and enforces type checking on the interface. In addition, this interface is based on a vendor-independent general multithreading package. Consequently, it provides a unified way to handle threads and synchronization regardless of the thread library being used.

The **liboo_asc** library includes the following classes:

- **ASC_Main** – Uses all the classes listed below in the library. It establishes and initializes a default operating environment for the application and then instantiates it. It also provides a pure virtual method **appl_initialize ()**, that is used to override the code written in the subclass. You must do the following:
  - subclass the **ASC_Main** class and write the application-specific code within member function **appl_initialize()**
  - write application specific cleanup code within **appl_cleanup()** if any
  - call **startup()** after instantiating the subclass in the core code
- **Diagnosis** – Provides a facility for logging diagnostic messages. It also maintains a single service thread for dequeueing diagnostic messages from a dedicated message queue and dumping them to the diagnostic files, if required.
- **Event** – Logs system events onto the database servers and diagnostic messages into the local diagnostic files. Event messages are queued on a dedicated event message queue waiting for pickup by the **EventAgent**.
- **EventAgent** – Has only one instance per application. It maintains a service thread to dequeue event messages from the event message queue and send **log_event** RPCs to the appropriate control server **ClientProc** objects that are managed by **ClntProcMgr** objects.
- **ClientProc** – Manages one connection to a specific server (for example, control server or database server) and sends RPCs to, and gets results back from, that server through the connection.
- **ClntProcMgr** – Manages multiple **ClientProc** objects (for example, multiple connections to a server). You can get the required **ClientProc** objects from appropriate **ClntProcMgr** objects using **getObj ()** method and return them using **returnObj ()** method.
- **Config** – Retrieves application configuration parameters set in the **ASAP.cfg** file. The first **Config** object instantiated from **ASC_Main** loads all the parameters into a binary tree, from which subsequent objects can retrieve what they need.
- **Common** – Utility class containing only static methods. These methods are used by the classes above and can be used anywhere if needed.

The following sections provide detailed definitions of the C++ classes in the **liboo_asc** library.

ASC_Main class

This class provides all the initialization methods required by ASAP applications, and it is the mainline front-end class of ASAP API. Application programmers must derive their own

subclass from this class and override the virtual method **ApplInit()** to provide application specific features, and then use the **startup()** method to start it.

The ASC_Main class has the following initialization methods:

- Configuration parameters initialization method **config_param_init**. This method loads the parameters from **ASAP.cfg** file and places them into an SBT.
- Sybase OpenClient library initialization method **ct_init**. This method does all the Sybase Open Client library related initialization actions.
- Input arguments processor **process_input**. This method processes the input parameters from the command line and places them into appropriate structures and variables.
- Signal handlers initialization methods **default_signal_handlers** and **install_signal_handler**. These methods install signal handlers and register these handlers to the Sybase Open Client library.
- Client application environment initialization method, **initialize**. This method:
  - Creates default event and diagnostic message queues.
  - Instantiates the first **Diagnosis** object and performs the initialization required. A diagnostic service thread is created at this stage if specified in the configuration file.
  - Retrieves server-related parameters.
  - Creates a default control database server connection manager (a **ClntProcMgr** object) that creates and manages a specified number of **ClientProc** objects connecting to the control database server.
  - Creates a default master control server connection manager (a **ClntProcMgr** object) that creates and manages a specified number of **ClientProc** objects connecting to the master control server.
  - If the slave control server differs from the master control server, this initialization method creates the default slave control server connection manager (a **ClntProcMgr** object) that creates and manages a specified number of **ClientProc** objects that connect to the slave control server.
  - Instantiates an **EventAgent** object that creates and manages an event service thread to wait for event messages on the default event message queue.

A pure virtual method **appl_initialize** is provided to enable application programmers to write their own application code in their subclasses.

If application programmers want to write custom cleanup code, they must define their own **appl_cleanup** method to override the **do nothing** behavior in the default method.

Synopsis

```
class ASC_Main: public Config, public ASC_ThreadAppl
{
public:
ASC_Main (int argc, char *argv[]);
~ASC_Main (void);
int threadMain(void **Rtn) { return CS_SUCCEED; };
CS_RETCODE config_param_init (void);
CS_RETCODE ctlib_init (void);
void process_input (void);
CS_RETCODE initialize (void);
```

```
void default_signal_handlers (void);
CS_RETCODE install_signal_handler (CS_INT signum, CS_VOID * func);
virtual CS_RETCODE appl_initialize (void) = 0;
virtual void appl_cleanup (void) {};
void startup(void);
protected:
int m_argc_;
// Local copy of main()'s argc.
char **m_argv_;
// Local copy of main()'s argv;
DIAG_CONFIG m_diag_cfg;
// Initialization structure use by class Diagnosis
// static Diagnosis *m_diag;
// Original diag object pointer.
EventAgent * m_event_ea_;
// EventAgent service thread object pointer.
char m_SQLSrvName[80];
// Physical SQL Server for the Control DB.
char m_MasterCtrlSvr[80];
// Logical Master Control Server.
static CS_CONTEXT *m_client_context_;
// Context of the client process.
// The following routines are Unix signal handlers.
// They are installed by Sybase's ct_callback()
// routine in default_signal_handlers() method.
// They can also be used as other signal's handler
// which should be installed separately by
// install_signal_handler() method.
static void terminate_signal(int sig);
static void core_dump_signal(int sig);
static void child_terminate(int sig);
static void ignore_signal(int sig);
static void terminate_pgm(void);
// routine used by signal handlers.
static ASC_Main *m_this_ptr;
// copy of this pointer.
};
```

Constructors

```
ASC_Main (int argc, char *argv[]);
```

This constructor uses all the initialization methods to establish the required client application environment. If the **ASC_Main** object instantiates successfully, the next step is to execute an application-specific code.

Arguments:

An **ASC_Main** object must be initialized with the following input arguments:

- **argc:** The number of elements in the **argv[ ]** array. You must pass the argc of **main ()** to this argument.
- **argv:** An array of string pointers. You pass the **argv** of **main ()** to this argument. Valid elements are:
  - **applName** – Application name
  - **-c ctrlSvrName** – Master control server name
  - **-l diagLevel** – Diagnostic level

- **-f diagFile** – Diagnostic file name

Public methods

The following are the public methods.

appl_initialize, appl_cleanup

You must override **appl_initialize** and **appl_cleanup** using the application-specific methods provided in the application programmers' subclasses. The overridden methods can use any facility provided by this class library and spawn as many threads as required, as long as they do not violate the rules.

Syntax:

```
virtual CS_RETCODE appl_initialize (void) = 0;
virtual void appl_cleanup (void) {};
```

startup

The **startup** method is used to start the execution of application code in **appl_initialize ()** that you have defined. After the **ASC_Main** object has been successfully instantiated, **startup ()** should be called to pass program control to the application.

Syntax:

```
void startup(void);
```

threadMain

Required by ASC_ThreadAppl as the thread start function.

Syntax:

```
int threadMain(void **Rtn) { return CS_SUCCEED; };
```

config_param_init

This method initializes the configuration B tree.

Syntax:

```
CS_RETCODE config_param_init (void);
```

ctlib_init

This method initializes client/server related config parameters.

Syntax:

```
CS_RETCODE ctlib_init (void);
```

process_input

This method processes the command line input parameters. It terminates the program if the input is improper.

Syntax:

```
void process_input (void);
```

initialize

Client initialization method.

Syntax:

```
CS_RETCODE initialize (void);
```

default_signal_handlers

This function installs the signal callback handlers.

Syntax:

```
void default_signal_handlers (void);
```

install_signal_handle

This method installs handler **func** for signal **signum**.

Syntax:

```
CS_RETCODE install_signal_handler (CS_INT signum, CS_VOID * func);
```

appl_initialize

This pure virtual function is the abstract method for application specific activities. Application programmers should override this function in their subclasses with their self-defined methods.

Syntax:

```
virtual CS_RETCODE appl_initialize (void) = 0;
```

appl_cleanup

This pure virtual function is the abstract method for application specific cleanup activities. Application programmers should override this function in their subclasses with their self-defined methods if there's any special cleanup to be performed at termination.

Syntax:

```
virtual void appl_cleanup (void) {};
```

Diagnosis class

This class provides methods for initializing the diagnostic environment, and generating diagnostic messages to a log file.

For more information on the ASAP Configuration file, see the *ASAP Server Configuration Guide*.

- **Direct Mode** – In the direct mode, an internal mutex is used in each Diagnosis object to synchronize the writings from multiple threads.

- **Indirect Mode** – All diagnostic messages are queued to a dedicated message queue, from which a dedicated thread takes these messages and logs them to the diagnostic file.

The name of the diagnostic file is specified with an **-f** prefix when initiating the process.

When you construct a Diagnostic object and generate diagnostic messages, you can create an object without an argument, or with the thread name, and then use the diag method to log diagnostic messages. No other action is required.

Internally, the following methods for logging messages are recommended:

- All instances of the **Diagnosis** class can write messages to the diagnostic files directly.
- All instances of the **Diagnosis** class can write to a dedicated diagnostic message queue with a no-wait option. A service thread is used to dequeue the messages and write them to the diagnostic files in this case.

If a service thread is required, the **Diagnosis** can create this thread and instruct it to wait on the message queue. Only one such service thread can be created if multiple instances of **Diagnosis** are used by the application. If the thread or the message queue do not work well, **Diagnosis** can switch to the first logging method.

To prevent inconsistent concurrent writings in a multithreaded environment, writing to the diagnostic files must be synchronized.

Synopsis

```
class Diagnosis: public Config, public ASC_ThreadAppl {
public:
Diagnosis (DIAG_CONFIG *cfg, ASC_MsgQueue *asc_msgq,

char *thrName);

// Constructor to initialize diagnostic environment with
// configuration structure **cfg**.
Diagnosis (char *thrName = "unknown"):ASC_ThreadAppl

(thrName) {};

// Do nothing constructor.
~Diagnosis(void);
virtual CS_RETCODE initialize (DIAG_CONFIG *cfg);
virtual void diag (VOIDPTR UNUSED(ptr), DIAG_LEVEL level,

char *type,int line, char *file, char *fmt

virtual void  diag_format (VOIDPTR ptr, DIAG_LEVEL

level,char *type, int line, char *file, void
(*format_fn)(FILE *outfile, VOIDPTR ptr)) {};

virtual void hex_dump (DIAG_LEVEL level, char *type,

int line, char *file,CS_BYTE *buf, int buf_len) {};

virtual void rpc_dump (DIAG_LEVEL level, int line,

char *file) {};
```

```
virtual void stack_trace (DIAG_LEVEL level, char *type,

int line, char *file) {};

void * service_mgr (void);
static CS_BOOL m_diag_queue;
static ASC_Mutex m_diag_mtx_;
int threadMain(void **Rtn);
protected:
void multi_diag_init (void);
//
CS_RETCODE print_queue (void);
// This method is used to take an message entry from
// the diag queue and print the associated
// message to the diag file.
CS_RETCODE create_srv_thread (void);
// This method creates a service thread used to process
// diag messages on the diag queue and print
// them to the diag file on a FIFO basis.
DIAG_LEVEL set_diag_level (char *level_msg);
char *set_diag_level_msg (DIAG_LEVEL level);
void pre_write_diag_file (char *file);
void post_write_diag_file (DIAG_LEVEL level,

char *file, long max_size);

void diag_file_open(char *filename);
void diag_file_move(char *filename);
void dump_configuration(void);
static FILE *m_diag_outfile;
private:
static int m_last_day;
static long m_log_file_max;
static DIAG_FILE *m_diag_files;
static CS_INT m_diag_idx;
static DIAG_LEVEL m_diag_level;
static DIAG_CONFIG m_diag_config;
static CS_BOOL m_diag_line_flush;
static CS_INT m_MAX_DIAG_FILES;
static ASC_MsgQueue *m_msgq;
// the pointer to the message queue object used for
// queueing all diag messages.
};
```

Constructors

This class has two constructors. To create the service thread, use the **attachThread** method in **ASC_ThreadAppl** class. To provide dequeueing and printing services, **service_mgr** is called in the thread context.

Syntax:

```
Diagnosis (DIAG_CONFIG *cfg, ASC_MsgQueue *asc_msgq,

char *thrName);

Diagnosis (char *thrName = "unknown"):ASC_ThreadAppl

(thrName){};
```

Arguments:

- **cfg:** Initializes **Diagnosis**-related global settings, diagnostic files, and spawn service thread if required.
- **asc_msgq:** Constructs custom **Diagnosis** objects. The newly-created object uses the settings, diagnostic files, and message queue initialized when instantiating the very first **Diagnosis** object in the **ASC Main**'s constructor.
- **thrName:** The thread Name in which the object is instantiated. It appears at the end of each diagnostic message indicating in which thread the message is generated.

Public methods

The following are the public methods.

diag

ASAP Application Server runtime diagnostic method **diag** is designed to generate diagnostic messages in the required format anywhere in the application.

Syntax:

```
virtual void diag (VOIDPTR UNUSED(ptr), DIAG_LEVEL level,

char *type, int line, char *file, char *fmt, ...);
```

Arguments:

- **ptr:** Only used by the debugger within the Interpreter.
- **level:** The diagnostic level of this particular function call.
- **type:** Character pointer identifying type or functional origin of the function call. Only the first 15 characters of this function call appear in the diagnostic file.
- **line:** The line number of the function in the source file, _ _**LINE**_ _.
- **file:** The file name from which the function was called, _ _**FILE**_ _.
- **fmt:** Character pointer to a **sprintf** format buffer containing the diagnostic message itself.
- **...:** Variable number of parameters following the **sprintf** format.

initialize

Initializes the diagnostic environment with configuration structure ****cfg****.

Syntax:

```
virtual CS_RETCODE initialize (DIAG_CONFIG *cfg);
```

diag_format

SAP Application Server User Specified Diagnostic Formatting method.

Syntax:

```
virtual void diag_format (VOIDPTR ptr, DIAG_LEVEL
```

```
level,char *type, int line, char *file, void
(*format_fn)(FILE *outfile, VOIDPTR ptr)) {};
```

hex_dump

This function produces a hexadecimal dump of a character buffer in the server diagnostic logfile.

Syntax:

```
virtual void  hex_dump (DIAG_LEVEL level, char *type,

int line, char *file,CS_BYTE *buf, int buf_len) {};
```

rpc_dump

This function prints the RPCs supported by the Open Server to the diagnostic logfile.

Syntax:

```
virtual void  rpc_dump (DIAG_LEVEL level, int line,

char *file) {};
```

stack_trace

```
virtual void  stack_trace (DIAG_LEVEL level, char *type, int line, char *file) {};
```

service_mgr

This member function is the start function of the queue service thread used to take diagnostic messages off the diag queue and print them to the diag file.

This method must only be used in conjunction with **create_srv_thread()** method and must not be used elsewhere.

Syntax:

```
void * service_mgr (void);
```

m_diag_queue

This flag indicates whether diag messages are sent to a message queue. If CS_TRUE is set, all messages are sent to the queue. If this flag is set to CS_FALSE after the service thread has been created, the service thread terminates after taking all messages off the queue and processing them.

Syntax:

```
static CS_BOOL m_diag_queue;
```

m_diag_mtx_

This method is a pointer to the mutex used to synchronize access to the diag file.

Syntax:

```
static ASC_Mutex m_diag_mtx_;
```

threadMain

ASC_ThreadAppl required thread start function.

Syntax:

```
int threadMain(void **Rtn);
```

Event class

This class provides the functionality to convert parameters into a system event and save it in the control database.

All **Event** objects log the event messages to the diagnostic file and queue them to a dedicated message queue. From this queue an EventAgent object (in another thread) takes the messages and logs them to the control server through RPC.

You can create multiple instances of the **Event** class and you can access different objects concurrently.

Whenever you generate an event through an **Event** object, the requested object:

- Logs a diagnostic message to the default diagnostic files.
- Converts your input parameters into an event message.
- Puts this event message onto the dedicated event message queue.

System event requests are not generated by **Event** objects. An **EventAgent** object dequeues the event messages from the dedicated event message queue and sends the log_event RPCs to the control server.

Synopsis

```
class Event: public Diagnosis
{
public:
Event (char * thrName ="Event"):

m_eventq_(default_eventq), Diagnosis(thrName) {};


void event(char *event_type, short line, char *file, char *fmt, ...);
// ASAP event logging method.
};
```

Constructors

The Event class uses a default event message queue to pass event messages to the **EventAgent** thread. A pointer to that default queue is recorded into data member **m_eventq_** when an object is constructed.

```
Event (char * thrName = "Event"):

m_eventq_(default_eventq), Diagnosis(thrName)
```

You pass the thread name of the creating thread to the constructor when you instantiate the Event object. This thread name appears at the end of the diagnostic

message logged into the diagnostic file. If no thread name is passed when instantiating an Event object, the event is taken as the name by default.

Arguments:

- **event_type:** Specifies the system event to be generated. This code is used to determine the operation to perform from the configuration tables.
- **line:** Line in source file where the system event was generated. Should be **__LINE__**.
- **file:** Source file where the system was generated. Should be **__FILE__**.
- **fmt: sprintf()** type format string specifying cause of the system event.
- **...:**Variable number of parameters following the **sprintf()** format.

Public methods

event:

ASAP event logging method.

Syntax:

```
void event(char *event_type, short line, char *file,

char *fmt, ...);
```

EventAgent class

The **EventAgent** class creates a service thread waiting on the dedicated event message queue for incoming event messages enqueued by **Event** objects. Each application process has only one such service thread.

If there is no message to be processed, this thread sleeps on the queue. When an event message comes, the service thread removes it from the queue, passes the information in the message structure to a ClientProc object, and then sends an RPC to the designated control server through the connection managed by that ClientProc object. If the RPC execution fails on the server side, a diagnostic message is written to the default diagnostic file. The ClientProc object is obtained from a ClntProcMgr object managing multiple ClientProc objects connected to the control server.

This class should only be instantiated once in an application and this unique instance should only be created in the constructor of ASC_Main. The attachThread () method from the class ASC_ThreadAppl is used in the constructor of the EventAgent class to create the service thread. The thread calls the start_service () method and waits for event messages on the default dedicated event message queue.

Synopsis

```
typedef struct {
int operation;
char event[APPL_EVENT_L+1];
char source_file[SOURCE_FILE_L+1];
short source_line;
char reason[APPL_REASON_L+1];
} CONTROL_AGENT_MSG;
class EventAgent: public ASC_ThreadAppl
{
```

```
public:
EventAgent (ASC_MsgQueue *eventq = ::default_eventq,

ClntProcMgr *sql_cpp = ::default_sql_cpp,
ClntProcMgr *ctrl_cpp = ::default_ctrl_cpp);

// This constructor initializes private data members,
// ~EventAgent (void) {};
int threadMain(void **Rtn);
// ASC_ThreadAppl required thread start routine.
void startup (void);
void start_service (void);
CS_BOOL m_should_terminate;
virtual void alarm (time_t seconds, ASC_MsgQueue *q,

int *msg, long msg_operation) {};

ASC_Mutex m_ea_mtx_;
protected:
void write_event (ClientProc *sql_cp, ClientProc

*ctrl_cp, CS_CHAR *event, CS_CHAR *file,
CS_INT *line, CS_CHAR *reason);

static CS_RETCODE sys_event_handler (CS_VOID *sql_cp,

CS_VOID *data,CS_RETCODE UNUSED(restype),
CS_BOOL *UNUSED(not_done));

private:
ASC_MsgQueue *m_eventq_;
// Message queue used to pass event messages to the
// agent thread.
ClntProcMgr *m_ctrl_cpp_;
// Pointer to master control server connections
// manager object.
ClntProcMgr *m_sql_cpp_;
// Pointer to control sql server connections
// manager object.
};
```

Constructors

```
EventAgent (ASC_MsgQueue *eventq = ::default_eventq,

ClntProcMgr *sql_cpp = ::default_sql_cpp,
ClntProcMgr *ctrl_cpp = ::default_ctrl_cpp);
```

This constructor initializes private data members, ~EventAgent (void) {};.

Public methods

The following are the public methods.

threadMain

ASC_ThreadAppl is a required thread to start the routine.

Syntax:

```
int threadMain(void **Rtn);
```

start_service

This method waits for event messages on the previously-created event message queue. If there is an incoming message, it generates a server event for the message.

Syntax:

```
void start_service (void);
```

m_should_terminate

This flag indicates whether the service thread should be terminated. If **CS_TRUE** is returned, the service thread must be terminated.

Syntax:

```
CS_BOOL m_should_terminate;
```

alarm

```
virtual void alarm (time_t seconds, ASC_MsgQueue *q,

int *msg, long msg_operation) {};
```

m_ea_mtx_

Mutex used to synchronize access to **should_terminate**.

Syntax:

```
ASC_Mutex m_ea_mtx_;
```

ClientProc class

The ClientProc class contains methods for client applications to handle connections and communications between clients and servers, and to send RPC execution requests to servers through these managed connections. A ClientProc object synchronously waits for the results of the RPC executions and invokes the appropriate handlers after it receives these results. It can be used safely in both one-thread-one-connection and multiple-threads-one-connection models.

Each object of this class is a client process context that manages a connection to a specific RDBMS Server, through which you can send RPCs to and receive results from the server. A client message handler and a server message handler, which log messages from the Sybase Open Client library and servers to a diagnostic file, are also installed internally upon creation of an object. In addition, concurrent accesses to the connection are synchronized inside an object so that the object can be used safely in both single and multiple thread connection models.

The ClientProc class encapsulates Sybase Open Client functionality and synchronization facilities to provide multithread-safe context allocation/drop, connection open/close, connection status check, and RPC execution/results binding methods to the application programmers. In addition, message handler methods for both the Sybase and Oracle Client Libraries and server feedbacks are provided in the class.

A ClientProc object can work in:

- **Dedicated mode** – Only accessed by a single thread (normally its creating thread) at any time.
- **Shared mode** – Accessed from multiple threads, in which case the connection to the server is shared. Access to a shared **ClientProc** object's server connection is serialized by the object itself.

Synopsis

```
ClientProc class:class ClientProc: public Diagnosis
{
public:
ClientProc (CS_CHAR *srv_name, CS_CHAR *userid,

CS_CHAR *password, CP_CONNECT_TYPE db_conn =
OPEN_SERVER);

ClientProc (CS_CHAR *srv_name, CS_CHAR *userid,

CS_CHAR *password, CS_VOID *data,
CLNT_HANDLER *hand_tbl, CM_RPC *rpcdef,
CP_CONNECT_TYPE db_conn, ...);

~ClientProc (void);
CS_RETCODE cpopen (void);
CS_RETCODE cpclose (void);
CS_RETCODE cpcheck (void);
CS_RETCODE cpbind (CS_INT base_idx, CM_RPC *rpcdef, ...);
CS_BOOL IS_OPEN (void);
CS_RETCODE  cprpcexec (CS_VOID *data, CLNT_HANDLER

*hand_tbl, CM_RPC *rpcdef, ...);

CLIENT_PROC * get_cp(void) { return m_cp_ptr_; }
CS_BOOL is_busy (void) { return m_is_busy_; }
CS_INT get_return_status (void){return m_return_status; }
CS_RETCODE cancelOperation(void);
CP_CONNECT_TYPE get_db_type(void) { return m_db_type_; }
static ASC_Mutex *m_alloc_mtx_;
static ASC_Mutex *m_init_mtx_;
static CS_RETCODE ct_msg_handler(CS_CONTEXT *cp,

CS_CONNECTION *chp,   CS_CLIENTMSG *msgp);

static CS_RETCODE srv_msg_handler(CS_CONTEXT *cp,

CS_CONNECTION *chp, CS_SERVERMSG *msgp);

#ifdef ORACLE_DB
//*********************************************************
// ORACLE database connection management public methods.
//
CS_INT ocicreate_list(Cda_Def *cda, ORA_COLUMN **colsptr,

CS_INT numrows);

CS_RETCODE ocidestroy_list(ORA_COLUMN *cols);
CS_RETCODE ocifetch(Cda_Def *cda, ORA_COLUMN *cols);
CS_RETCODE ocistatus(Cda_Def *cda);
```

```
CS_RETCODE ocican_cursor(Cda_Def *cda);
CS_RETCODE ociopen_cursor(void);
CS_RETCODE ociclose_cursor(Cda_Def *cda);
CS_RETCODE ociparse(CS_CHAR *command);
CS_RETCODE ocicreate_cmd(CS_CHAR *command,

CM_RPC *rpcdef);

CS_RETCODE ocirpcexec(CM_RPC * rpcdef);
#endif
void dts_to_str(CS_CHAR *date_str, CS_DATETIME *dts);
void str_to_dts(CS_CHAR *date_str, CS_DATETIME *dts);
private:
CLIENT_PROC * m_cp_ptr_;
// The unique client process structure used
// within the object.
ASC_Mutex m_glob_sync_mtx_;
// ASC_Mutex used to synchronize the access to the
// object if this object is shared by multiple threads.
virtual CS_RETCODE
process_status(CM_RPC *rpcdef, CS_BOOL *not_done,

CS_BOOL *rpc_error, CS_BOOL *deadlock_error,
CS_INT retry_num);

virtual CS_RETCODE
initialize_ct_context (CS_CONTEXT *cp,

CS_RETCODE (*clientmsg_cb)(CS_CONTEXT *cp,
CS_CONNECTION *conn,
CS_CLIENTMSG *msg),
CS_RETCODE (*servermsg_cb)(CS_CONTEXT *cp,
CS_CONNECTION *conn,
CS_SERVERMSG *msg)
;

// ct library context initialization method.
static CS_RETCODE ct_msg_handler(CS_CONTEXT *cp,

CS_CONNECTION *chp,
CS_CLIENTMSG *msgp);

static CS_RETCODE srv_msg_handler(CS_CONTEXT *cp,

CS_CONNECTION *chp,
CS_SERVERMSG *msgp);

protected:
CS_BOOL m_is_busy_;
// This flag is used to check if this object is
// in use (busy);
// CS_TRUE is set if busy.
CS_BOOL m_is_connected_;
// This flag is used to check if this object is
// connected to the server;
// CS_TRUE is set if connected.
};
```

Constructors

This constructor allocates and initializes CLIENT_PROC structure, and opens connections to the server. If it is used to instantiate an object, a connection to the server specified by the input argument srv_name is established by default. RPC execution method cprpcexec () can be used right after the object initialization. You can use it to open dedicated connections to servers.

Syntax:

```
ClientProc (CS_CHAR *srv_name, CS_CHAR *userid,

CS_CHAR *password, CP_CONNECT_TYPE db_conn =
OPEN_SERVER);
```

This constructor allocates and initializes CLIENT_PROC structure, opens connections to the server, executes an RPC registered procedure on the server and then processes the results. If this constructor is used to instantiate an object, a connection to the server specified by the input argument **srv_name** is established, an RPC execution request is sent to the server and the results are processed. You can use this constructor to execute a single RPC on the server. This constructor initializes its parent class **Diagnosis**. This is required when an object is created for use by multiple threads. A different thread can open a connection to different servers or send RPCs to different destinations.

Syntax:

```
ClientProc (CS_CHAR *srv_name, CS_CHAR *userid,

CS_CHAR *password, CS_VOID *data,
CLNT_HANDLER *hand_tbl, CM_RPC *rpcdef,
CP_CONNECT_TYPE db_conn, ...);
```

Arguments:

- **db_conn:** Indicates the connection type, and is one of the following:
  - OPEN_SERVER – Sybase Open Server connection.
  - ORACLE – Oracle Server RDBMS connection.

Public methods

This section lists the public methods.

cprpcexec

```
CS_RETCODE  cprpcexec (CS_VOID *data, CLNT_HANDLER

*hand_tbl, CM_RPC *rpcdef, ...);
```

This method executes an RPC or a registered procedure on the server and then processes the results. The returned results are mapped and then the appropriate result handler is called to process the data.

Arguments:

- **data:** Generic data pointer that is passed to the handlers once the results are returned by the server.
- **hand_tbl:** Result handler table to process the return data.
- **rpcdef:** RPC definition structure specifying the RPC to be executed and its associated parameters.
- **...:** Variable parameter list specifying the values for the RPC parameters identified by **rpcdef**. All parameters must be pointers.

Return values:

- **CS_SUCCEED:** RPC execution was successful and the results were handled successfully.
- **CS_FAIL:** The function failed due to an RPC execution error or one of the result handlers failed.

cpopen

This method opens a connection to the server that is specified by private member **cp_ptr_**, which is returned by **cpalloc** method. **CS_SUCCEED** is returned if successful.

Syntax:

```
CS_RETCODE cpopen (void);
```

cpclose

This method closes the object's connection to the server. **CS_SUCCEED** is returned if successful.

Syntax:

```
CS_RETCODE cpclose (void);
```

cpcheck

This method checks the network connection to verify that the server is available. When an error is detected, the connection is automatically closed if it is open.

Syntax:

```
CS_RETCODE cpcheck (void);
```

IS_OPEN

This method checks if the current connection associated with the object is still open.

Syntax:

```
CS_BOOL IS_OPEN (void);
```

get_cp

Return the unique client process structure pointer.

Syntax:

```
CLIENT_PROC * get_cp(void) { return m_cp_ptr_; }
```

is_busy

Check if the object is busy.

Syntax:

```
CS_BOOL is_busy (void) { return m_is_busy_; }
```

get_return_status

Retrieves the RPC status.

Syntax:

```
CS_INT get_return_status (void){return m_return_status; }
```

cancelOperation

Cancels the current database operation.

Syntax:

```
CS_RETCODE cancelOperation(void);
```

get_db_type

Returns OPEN_SERVER (0) if connection is Open Server. Returns SYBASE (1) if connection is Sybase SQL Server. Returns ORACLE (2) if connection is Oracle Server.

Syntax:

```
CP_CONNECT_TYPE get_db_type(void) { return m_db_type_; }
```

m_alloc_mtx_

The ASC_Mutex used to synchronize context allocations and droppings.

Syntax:

```
static ASC_Mutex *m_alloc_mtx_;
```

m_init_mtx_

ASC_Mutex is used to synchronize the Open Client library **ct_init()** and **ct_exit()** function calls.

Syntax:

```
static ASC_Mutex *m_init_mtx_;
```

Public methods

The following are the ORACLE database connection management public methods.

ocicreate_list

This function uses the **odescr** function to determine the columns name, data type and data size for every column in the cursor variable.

Syntax:

```
CS_INT ocicreate_list(Cda_Def *cda, ORA_COLUMN **colsptr, CS_INT numrows);
```

ocidestroy_list

Frees storage allocated to ORA_COLUMN structure.

Syntax:

```
CS_RETCODE ocidestroy_list(ORA_COLUMN *cols);
```

ocifetch

This function attempts to fetch as many rows as specified in **numrows**.

Syntax:

```
CS_RETCODE ocifetch(Cda_Def *cda, ORA_COLUMN *cols);
```

ocistatus

Checks the status of **Cda** of CLIENT_PROC.

Syntax:

```
CS_RETCODE ocistatus(Cda_Def *cda);
```

ocican_cursor

Cancels a query on a cursor after the required rows have been fetched.

Syntax:

```
CS_RETCODE ocican_cursor(Cda_Def *cda);
```

ociopen_cursor

This function associates a cursor data area in the application with a data area in the Oracle server. These are used by Oracle to maintain state information about the processing of a SQL statement.

Syntax:

```
CS_RETCODE ociopen_cursor(void);
```

ociclose_cursor

This function disconnects a cursor from the data areas in the Oracle Server. Because you can associate more than one cursor with an RPC (cursor variable processing), you cannot associate all of them with a single CLIENT_PROC. Therefore, this function only takes **Cda** as an argument.

Syntax:

```
CS_RETCODE ociclose_cursor(Cda_Def *cda);
```

ociparse

This function parses a SQL statement and associates it with a cursor. For performance reasons, all parsing is done in deferred mode. Therefore, PLSQL syntax errors are not likely to be detected until after the RPC is executed.

Syntax:

```
CS_RETCODE ociparse(CS_CHAR *command);
```

ocicreate_cmd

This function builds a PL/SQL call based on the CM_RPC structure. The resulting command can be passed to ociparse to be associated with an open cursor. To receive a return status from an RPC to the Oracle server functions are used on the server side instead of procedures.

Syntax:

```
CS_RETCODE ocicreate_cmd(CS_CHAR *command, CM_RPC *rpcdef);
```

ocirpcexec

This function invokes the RPC associated with the specified cursor to the Oracle Server.

Syntax:

```
CS_RETCODE ocirpcexec(CM_RPC * rpcdef);
```

dts_to_str

```
void dts_to_str(CS_CHAR *date_str, CS_DATETIME *dts);
```

Converts the SYBASE datetime to string date time format.

str_to_dts

```
void str_to_dts(CS_CHAR *date_str, CS_DATETIME *dts);
```

Converts the string date time format to SYBASE datetime.

MT-Safety in shared mode

To ensure multithread safety in the shared mode, access to a shared **ClientProc** object must be serialized. This is enforced by **lock_object ()** and **trylock_object ()** methods.

Before using other methods in the class, a thread must call **lock_object ()** or **trylock_object ()** to lock the object. If the object is already in use by another thread, **lock_object ()** is used. If the object is not free, **trylock_object ()** is used. After using the object, you must call **unlock_object ()** to enable another thread to lock it.

ClntProcMgr class

An object of this class is designed to manage all active ClientProc objects (for example, shared server connections). These ClientProc objects are connected to the same server. It maintains a list of CLNT_SVR_ST nodes for available objects. You can obtain a required ClientProc object from the ClntProcMgr object. When the object is no longer required, you can return it to the ClntProcMgr.

To replace the ClientProc object, you can call the member function replaceBadObj() directly (for example, if the object server connection is broken).

A ClntProcMgr object must never be deleted unless you are sure that no other threads are accessing this object.

The ClntProcMgr class manages grouped ClientProc objects. Each instance of the ClntProcMgr class manages a group of ClientProc objects initially connected to the same server. It creates newly managed ClientProc objects after its instantiation, if required, using the createObj method. Upon destruction, all ClientProc objects managed are deleted.

You can use the getObj method to get a pointer to a free ClientProc object for use, and use the returnObj method to return that ClientProc object for recycling.

The number of all ClientProc objects managed by an instance and the number of free ones are tracked. You use the checkNumOfFreeObj method to check the number of free objects available for use.

Synopsis

```
class ClntProcMgr
{
public:
ClntProcMgr (char *name, CS_CHAR *srv_name, CS_CHAR

*userid, CS_CHAR *password, CS_INT size,
CP_CONNECT_TYPE db_conn);

~ClntProcMgr (void);
//Class Destructor
CS_RETCODE createObj (void);
void deleteAllObj (void);
ClientProc *getObj (void);
void returnObj (ClientProc *obj);
ClientProc *replaceBadObj(ClientProc *badObj);
CS_INT checkNumOfFreeObj (void);
private:
ASC_Mutex m_conn_mtx_;
// Mutex used by objects of this class to provide
// consistent operations on the following static
// private data members.
CS_INT m_num_of_free_;
// Number of free objects for use.
CS_INT m_num_of_obj_;v// Number of all ClientProc objects managed by this.
CLNT_SVR_ST m_list_head;
// Head of the free objects list.
char m_ObjName[80];
// The name of this ClntProcMgr object.
};
```

Constructors

Only one type of constructor is provided in the class. This constructor calls the **createObj** method to create (size) **ClientProc** objects, allocate and initialize CLIENT_PROC structures, and open connections to the server specified by the input argument **srv_name**. The argument name specifies the name of the **ClntProcMgr** object, which you use when you search it from a group of **ClntProcMgr** objects.

Syntax:

```
ClntProcMgr (char *name, CS_CHAR *srv_name, CS_CHAR

*userid, CS_CHAR *password, CS_INT size,
CP_CONNECT_TYPE db_conn);
```

Arguments:

- **db_conn:** Indicates the connection type, and is one of the following:
  - OPEN_SERVER – Sybase Open Server connection.
  - ORACLE – Oracle Server RDBMS connection.

Public Methods

The following are the public methods.

createObj

```
CS_RETCODE createObj (void);
```

This method creates a **ClientProc** object, allocates and initializes CLIENT_PROC structures, and opens connections to the server.

deleteAllObj

```
void deleteAllObj (void);
```

This method destroys all free **ClientProc** objects managed by the class.

getObj

```
ClientProc *getObj (void);
```

This method is used to get a free **ClientProc** object pointer from the **ClntProcMgr** object. If no **ClientProc** object is available, a NULL is returned. You must check the return value of this method before using the pointer, or check the number of free **ClientProc** objects available using **checkNumOfFreeObj** method before making a call to **getObj ()**.

returnObj

```
void returnObj (ClientProc *obj);
```

Returns **ClientProc** object for recycling. This method puts this object back on the available objects list. You must ensure each **ClientProc** is returned after being used to maximize the usability of this group of **ClientProc** objects.

replaceBadObj

```
ClientProc *replaceBadObj(ClientProc *badObj);
```

Passes the bad **ClientProc** object back and gets a new one.

checkNumOfFreeObj

```
CS_INT checkNumOfFreeObj (void);
```

Returns the number of available objects on the object list.

Config class

This utility class retrieves system configuration parameters originally stored in the ASAP.cfg configuration file. Because public methods are static, you can use them directly with the **Config::** prefix.

The **Config** class retrieves configuration parameters initially defined in the ASAP.cfg file. All parameters in the file are loaded and put into a self-balancing tree (SBT) by the **ASC_Main** constructor that initializes the application environment.

Each parameter is identified by a unique name. This class provides the **get_config_param** method to search the value of only one parameter per use. If the parameter does not have an entry in the self-balancing tree, an entry is inserted with the default value provided.

As concurrent access to the SBT is possible in a multithreaded environment, this class uses a mutex facility to synchronize writing (adding new node) efforts from different threads.

Use the **dump_config_params** method to dump all the parameters to a specified file for diagnostic purposes.

Synopsis

```
class Config
{
public:
Config (void) {};
//Class destructor.
~Config (void) {};
static void  get_config_param (char *param,

char *value, char *default_vlu);

//
static void  dump_config_params (FILE *fp);
//
static int node_compare(const void *node1,

const void *node2);

//
static void dump_param_dtls(const void *node,
```

```
VISIT order, int level);

//
protected:
CS_RETCODE config_param_init (void);
// This method will initialize the configuration
// parameter B tree with the parameter present
// in both the global and application specific
// configuration files.
// get_config_param related methods below:
static void process_file(FILE *fp, char *file);
//
static char *insert_updt_node(char *lbl, char *vlu,

char *file, CS_BOOL initialization);

//
static void null_end_pad(char *string);
//
private:
static FILE *m_diag_fp;
static ASC_Mutex m_conf_mtx_;
// the mutex used to synchronize cfg related operations.
};
```

Constructors

No parameter is required when instantiating an object. A global self-balancing tree structure is created in the **ASC_Main** constructor for use by all Config objects.

Syntax:

```
Config (void) {};
```

Public methods

The following are the public methods.

get_config_param

If this entry is not present as either an application or global configuration parameter, then the default value is returned and it is set as a new entry. This method references the configuration file and returns the value of the requested parameter.

Syntax:

```
static void get_config_param (char *param, char *value, char *default_vlu);
```

Arguments:

- **param:** The parameter name.
- **value:** Character pointer to the value of the requested parameter. The returned parameter is placed in this location.
- **default_vlu:** Character pointer to the default value to be used for this parameter if the parameter is not listed in the configuration file.

dump_config_params

```
static void dump_config_params (FILE *fp);
```

This method dumps all parameters listed in the SBT to a file specified by a file descriptor **fp**.

Common class

The Common class is designed to provide some commonly used general purpose facilities on a platform independent basis. Currently it only provides time-related methods to get the current time and date.

Synopsis

```
class Common
{
public:

Common (void) {};
//Class constructor.
~Common (void) {};
//Class destructor.
static CS_RETCODE curDts(CS_DATETIME &dts);
static CS_FLOAT cur_tm(void);
static int today(void);

};
```

Constructors

Only one type of constructor is provided in this class.

Syntax:

```
Common (void) {};
```

Public Methods

The following are the public methods.

curDts

Gets the current date and time and places them in the dts structure.

Syntax:

```
static CS_RETCODE curDts(CS_DATETIME &dts);
```

cur_tm

Returns the current time.

Syntax:

```
static CS_FLOAT cur_tm(void);
```

today

Returns the current date.

Syntax:

```
static int today(void);
```

ASC thread library

Thread Classes are the core of the thread framework. They provide a generic interface to different thread architectures. Thread Classes consist of generic thread interface classes and vendor-specific classes.

The thread framework consists of the following classes:

- **ASC_Thread** – Provides a plain thread which has basic thread functions. The vendor-specific thread class, for example, **DCE_Thread**, is derived from this class.
- **ASC_ThreadFactory** – Has knowledge of instantiating a class.
- **ASC_ThreadAppl** – The application **Appl Class** is derived from this class. This class provides the interfaces for the application to access the thread factor, the thread, and the messaging system. After you have generated the **Appl Class** and attached it to a plain thread, the thread starts to execute the application you have defined.
- **Generic Message Classes** – Provides the application a generic interface to different messaging systems such as interthread communication (ITC), interprocess communication (IPC) etc. ThreadMsgQueue is ITC.

ASC_Thread class

This is an abstract class. Its member functions provide a generic interface to a thread, for example, detaching, joining, or exiting a thread. Since not all features are supported by all thread packages, the **isSupported** function reports when a feature is available.

Synopsis

```
class ASC_Thread {
public:

    ASC_Thread(void){}
    ~ASC_Thread(void){}
    virtual void exitThread(void *RtnData) = 0;
    virtual int detachSelf(void) = 0;
    virtual int detachThread(const ASC_Thread *ThreadObj) = 0;
    virtual void yieldThread(void) = 0;
    virtual int joinThread(const ASC_Thread *ThreadObj, void **RtnData) = 0;
    virtual int threadSigWait(void) = 0;
    virtual int operator==(const ASC_Thread *ThreadObj) = 0;
    virtual int isSupported(int ThreadMethod) = 0;

protected:
private:
};
```

Public methods

A vendor-specific thread class must subclass from ASC_Thread to interface with the vendor-specific thread functions.

The thread factory uses **spawnThread** to start a thread. The function interfaces with the vendor-specific thread function and physically spawns a thread. To attach the object to the thread, **ApplObj** is passed to the function.

ASC_ThreadFactory class

ASC_ThreadFactory is an abstract class. It provides a generic interface to start a thread. Each class library of the vendor-specific thread, has its own factory class derived from this class. The derived class contains information on how to instantiate a thread object. The object instantiated from this class is a singleton object.

Synopsis

```
class ASC_ThreadFactory {
public:

    ASC_ThreadFactory(void) {}
    ~ASC_ThreadFactory(void);
    ASC_Thread* createThread(ASC_ThreadAppl *ApplObj);
    virtual ASC_Mutex* createMutex(void) = 0;
    virtual ASC_Condition* createCondition(void) = 0;
    virtual void deleteMutex(ASC_Mutex *TheMutex) = 0;
    virtual void deleteCondition(ASC_Condition *TheCondition) = 0;
    virtual ASC_Mutex *getFactoryMutex(void) = 0;
    int getNumThreads(void) const {return m_ThreadInfoByID.size(); }
    const ASC_Thread *getThreadObj(const char *ThreadName);
    void threadTermination(ASC_ThreadAppl *ThreadAppl, int isExit,
        void* ReturnData);
    void ThreadIsJoined(ASC_Thread* TheThread);
    inline std::vector<ThreadInfo*>::iterator
    find_by_threadName(std::vector<ThreadInfo*> &container, const char *threadName);
    inline std::vector<ThreadInfo*>::iterator
    find_by_threadObj(std::vector<ThreadInfo*> &container, const ASC_Thread *obj);

protected:
    virtual ASC_Thread *spawnThread(ASC_ThreadAppl *ApplObj) = 0;
    virtual void threadCleanUp(const ASC_ExitInfo &Info) = 0;

private:
    std::vector<ThreadInfo*> m_ThreadInfoByID;
    ThreadInfo* removeThreadInfo(const ASC_Thread* Thread);
    ASC_ThreadFactory(const ASC_ThreadFactory &TheThreadFactory);
    ASC_ThreadFactory & operator=(const ASC_ThreadFactory &TheThreadFactory);
};
```

Public methods

To start a thread, **createThread** is invoked by the application. This function, in turn, invokes **spawnThread** to instantiate a thread object and start a thread. In the framework, createThread is called by **ASC_ThreadAppl**.

This class also saves the thread information, such as thread identifiers, pointers to thread objects, and thread names. When a thread is generated, the information is stored. The information is not used now and can be provided for future implementation, for example, to query and access a thread from outside of the process. When a thread is terminated, the information is discarded. Specifically, **createThread** saves information and **threadTermination** removes the information. When it terminates, **threadTermination** is invoked by a thread object.

A mutex object is used to ensure the synchronization of starting threads, since the factory can be used by more than one thread at the same time.

ASC_ThreadAppl class

This is the most important class for those who use the thread framework. Although an understanding of **ASC_Thread** and **ASC_ThreadFactory** is not necessary, you must understand this class.

This is an abstract class that the application must derive from. This class provides all the required interface methods for the application to start a thread, attach the application to the thread, and use the messaging system.

You use this class to generate an application running in a thread.

The subclass from this class is:

```
class Appl: public ASC_ThreadAppl
{
public:

Appl(char *ApplName):ASC_ThreadAppl(ApplName){}
~Appl(void){}
// This is a pure virtual and you must refine it.
// After the thread is started, this function is
// executed automatically.
int threadMain(void **Rtn);
};
// This thread application does nothing, but sending
// a message through a message queue again and again.
int Appl1:: threadMain(void **Rtn);
{
ASC_MsgQueue *TheQueue;
int QID, i = 0;
char *MsgText;
ASC_Msg *Msg;

// Generate a message queue with name Appl using
// the member function of ASC_ThreadAppl.

TheQueue = genMsgQueue("Appl", QID);
printf("This is Appl1 with Queue %d ThreadID
%d\n", QID, getThreadID());

while(1){

MsgText = new char[80];
sprintf(MsgText, "This is message $d", ++i);
Msg = new ASC_Msg(0, MsgText);

// Send a message using the member function
// of ASC_MsgQueue
```

```
TheQueue->putMsg(Msg, getThreadID(), 0);
}
return 0;

}
```

Instantiate the application object and attach it to a thread in **main()**.

```
main(int argc, char *argv[])
{
// Create a thread object with name Appl

Appl *a1 = new Appl("Appl");

// Attach it to a thread. After this call, the
// threadMain function will be executed in the
// thread and all subsequent calls.

a1->attachThread();
...

}
```

This class must provide all required interfaces for the application to use the thread framework. This class hides the detail how to:

- Generate a thread through the thread factory.
- Attach the application object to the thread.
- Obtain a message queue from the queue manager.

This class also provides functions for using methods provided by the thread it attached, retrieving some thread information, terminating the attached thread, etc.

The following is the class definition.

Synopsis

```
class ASC_ThreadAppl {
public:

ASC_ThreadAppl(char *ThreadName)
:m_TheThread(0), m_QueueList(0), m_Autodelete(0),
m_ThreadStarted(1)
{ m_ThreadName = new char[strlen(ThreadName) + 1];
strcpy(m_ThreadName, ThreadName); }
virtual ~ASC_ThreadAppl(void);
void attachThread(void);
int terminateThread(void *Rtn);
virtual int threadMain(void **Rtn) = 0;
char *getThreadName(void) { return m_ThreadName; }
int getThreadID(void);
const ASC_ThreadAttr & getThreadAttr(void) {return
m_TheAttr; }
void initAttr(void)
void setAttr(int AttrID, in Attribute);
ASC_MsgQueue* genMsgQueue(char *QueueName, int &QueueID);
ASC_MsgQueue* getMsgQueue(int QueueID);
ASC_MsgQueue* getMsgQueue(char *QueueName);
void delMsgQueue(char *QueueName);
void delMsgQueue(int *QueueID);
```

```
void delMsgQueue(ASC_MsgQueue* MsgQueue);
void setTheQueue(ASC_MsgQueue* MsgQueue, int QueueID);
ASC_MsgQueue* getTheQueue(int QueueID);
void setTheThread(ASC_Thread *TheThread)
{ m_TheThread = TheThread; }
ASC_Thread* getTheThread(void) { return m_TheThread; }
void theThreadTerminated(void) { m_TheThread = 0; }
void setAutoDelete(void) { m_AutoDelete = 1; }
int AutoDelete(void) { return m_AutoDelete; }
ASC_Mutex & getTheMutex(void) { return m_TheMutex; }
int threadStarted(void) { return m_ThreadStarted; }

private:

char *m_ThreadName;
ASC_Thread *m_TheThread;
ASC_ThreadAttr m_TheAttr;
MsgQueueList *m_QueueList;
int m_AutoDelete;
int m_ThreadStarted;
ASC_Mutex m_TheMutex;

};
```

Public methods

The following are the public methods.

attachThread

This function creates a new thread and attaches the application object to it. The implementation protects the race condition, where the attached thread exits before **attachThread** is completed. This is achieved using a mutex which attaches to the application object.

Syntax:

```
void attachThread(void);
```

terminateThread

The application uses **terminateThread** to terminate the attached thread. To use this method to terminate a thread, the application must not detach the thread. This method uses a dedicated daemon thread and synchronous message to ensure the application object is deleted after the thread is terminated.

Syntax:

```
int terminateThread(void *Rtn);
```

threadMain

After an application object is attached to a thread, **threadMain** is executed. All subsequent calls to the member functions and other functions are executed in the thread. The thread is terminated when **threadMain** is exited. The application can exit **threadMain** either through a **return** statement or by invoking the member function **terminateThread**.

If the application calls **autoDelete**, the application object is deleted when the thread is terminated, otherwise, the application object is kept intact after the thread is terminated.

Syntax:

```
virtual int threadMain(void **Rtn) = 0;
```

getThreadName, getThreadID

This function retrieves the thread name.

Syntax:

```
char *getThreadName(void) { return m_ThreadName; }
int getThreadID(void);
```

ASC_ThreadAttr, getThreadAttr

Gets the attribute object.

Syntax:

```
const ASC_ThreadAttr & getThreadAttr(void) {return m_TheAttr; }
```

initAttr, setAttr

Initializes the attribute object. **initAttr** should be called once before subsequent calls to **setAttr()**.

Syntax:

```
void initAttr(void)
void setAttr(int AttrID, in Attribute);
```

genMsgQueue

Creates a message queue.

Syntax:

```
ASC_MsgQueue* genMsgQueue(char *QueueName, int &QueueID);
```

getMsgQueue

Gets a message queue from the session manager. For performance reasons, the application should keep the pointer to the message queue(s) it uses, instead of using **getMsgQueue** to retrieve it from the queue manager each time.

Syntax:

```
ASC_MsgQueue* getMsgQueue(int QueueID);
ASC_MsgQueue* getMsgQueue(char *QueueName);
```

delMsgQueue

Deletes a message by name or ID.

Syntax:

```
void delMsgQueue(char *QueueName);
void delMsgQueue(int *QueueID);
void delMsgQueue(ASC_MsgQueue* MsgQueue);
```

setTheQueue

Saves the message queue locally.

Syntax:

```
void setTheQueue(ASC_MsgQueue* MsgQueue, int QueueID);
```

getTheQueue

Gets a local message queue by ID.

Syntax:

```
ASC_MsgQueue* getTheQueue(int QueueID);
```

setTheThread

Assigns the pointer to the thread to **m_TheThread**.

Syntax:

```
void setTheThread(ASC_Thread *TheThread){ m_TheThread = TheThread; }
```

DCE_Thread class

The DCE thread is typically used when implementing a vendor-specific thread library. This library is linked with the generic thread class library only when the application chooses to use the DCE threads. Another vendor-specific thread library is linked otherwise.

This DCE Thread Class library contains the **DCE_Thread** and **DCE_ThreadFactory** classes.

DCE_Thread is a subclass of **ASC_Thread**. This class redefines all methods defined in **ASC_thread** as pure virtual functions. These methods provide a generic interface for the application to use the thread functions, provided by DCE thread vendors.

Synopsis

```
class DCE_Thread : public ASC_Thread {
public:

DCE_Thread(void):m_ThreadID(-1),m_detached(0),
m_SigSet(0),m_ApplObj(0){}~DCE_Thread(void){}
void spawnThread(int &ThreadID, ASC_ThreadAppl *ApplObj);
void exitThread(void *RtnData);
int detachSelf(void);
int detachThread(int ThreadID);
void yieldThread(void);
int joinThread(int ThreadID, void **RtnData);
int threadSigWait(void);
int setSigSet(ASC_SigSet *SigSet);
int isSupported(int ThreadMethod){ return ASC_True; }
```

```
// This function retrieves thread ID
int getThreadID(void) { return (int)m_ThreadID; }

// This function converts generic attributes to
// DCE thread attributes.
pthread_attr_t getAttributes(ASC_ThreadAppl *ApplAttr);
;
```

Public methods

The following are the public methods.

spawnThread

Spawns a DCE thread.

Syntax:

```
void spawnThread(int &ThreadID, ASC_ThreadAppl *ApplObj);
```

isSupported

This function retrieves thread ID **int getThreadID(void) { return (int)m_ThreadID; }**. To determine whether a method is supported by a particular thread package, use **isSupported**. Returning **ASC_True** identifies that the method is supported. DCE threads support all methods; therefore, it returns **True** all the time.

Syntax:

```
int isSupported(int ThreadMethod){ return ASC_True; }
```

getAttributes

This function converts generic attributes to DCE thread attributes.

Syntax:

```
pthread_attr_t getAttributes(ASC_ThreadAppl *ApplAttr);
```

DCE_ThreadFactory class

DCE_ThreadFactory class is a subclass of **ASC_ThreadFactory**. This class mainly defines the spawnThread method because only the DCE thread factory can create a DCE thread object. This method instantiates the DCE thread object, and then invokes its method, **spawnThread**, to spawn a DCE thread.

Synopsis

```
class DCE_ThreadFactory : public ASC_ThreadFactory {
public:

DCE_ThreadFactory(void){}
~DCE_ThreadFactory(void){}
ASC_Thread *spawnThread(int &ThreadID, ASC_ThreadAppl
*ApplObj);
```

```
protected:
};
```

Public method

The following are the public methods.

spawnThread

Spawns a DCE thread.

Syntax:

```
void spawnThread(int &ThreadID, ASC_ThreadAppl *ApplObj);
```

DCE implementation

In addition to the DCE classes mentioned above, there are DCE-related implementations. The mutex and condition functions are defined in **ASC_Mutex** class, however, the implementation of the methods are DCE specific.

The mutex is used by both the framework and the application. The methods of the mutex class provide a generic interface to use vendor specific mutex and conditions. The following is a mutex class.

ASC_Mutex class

You can implement these methods in the DCE library to interface with vendor-specific thread packages. You can also implement the mutex by subclassing **ASC_Mutex** in the DCE library and to request a mutex from **DCE_ThreadFactory**. In this way, a mutex must be implemented in the application as an object.

Synopsis

```
class ASC_Mutex {
public:

ASC_Mutex(void);
~ASC_Mutex(void);
ASC_Mutex & operator=(const ASC_Mutex &);
ASC_Mutex(const ASC_Mutex &);
int lock(void);
int unlock(void);
int tryLock(void);
int condWait(void);
int condTimeWait(float Timeout);
int condSignal(void);
int condBroadCast(void);
int isSupported(int MethodID);

private:

void *m_MutexData;

};
```

Constructors

```
ASC_Mutex(void);
~ASC_Mutex(void);
```

This constructor initializes the mutex and condition variables.

```
ASC_Mutex & operator=(const ASC_Mutex &);
```

This is the assignment constructor.

```
ASC_Mutex(const ASC_Mutex &);
```

This is the Copy constructor.

Public methods

The following are the public methods.

lock, unlock

These functions lock and unlock the mutex.

Syntax:

```
int lock(void);
int unlock(void);
```

trylock

This function tries to lock the mutex. If it cannot, it returns instead of blocking the caller.

Syntax:

```
int tryLock(void);
```

condWait

This function waits for a condition to become true.

Syntax:

```
int condWait(void);
```

condTimeWait

This function waits for a condition to become true during a period of time specified by Timeout.

Syntax:

```
int condTimeWait(float Timeout);
```

condSignal

Sends a signal when a condition is changed.

Syntax:

```
int condSignal(void);
```

condBroadCast

Broadcasts a signal when a condition is changed.

Syntax:

```
int condBroadCast(void);
```

isSupported

Use this function to determine whether a method is supported.

Syntax:

```
int isSupported(int MethodID);
```

ASC_Context class

The mutex and condition are implemented as two classes instead of one.

In this framework, a context class is introduced for the application to retrieve the thread factory and message manager.

The ASC_Context class can be implemented in the following ways.

- Similar to the implementation of the mutex, the **ASC_Context** is implemented in the DCE to instantiate the DCE thread factory. In this way, the application is not required to choose the thread factory at run time, but at linking time.
- Another way to implement this class is to let the application choose the thread factory at run and link time. In this approach, the context class is not necessary.

The first approach is not considered proper in a C++ implementation, because it exposes the class private members to other libraries.

Synopsis

```
class ASC_Context {
public:

~ASC_Context(void){}
static ASC_Context *getTheContext(void);
ASC_ThreadFactory *getThreadFactory(void);
int deleteThreadFactory(void);
MsgQueueMgr *getMsgQueueMgr(void);
int deleteMsgQueueMgr(void);

protected:

// These routines must be created in platform
// dependent thread libraries.
ASC_ThreadFactory *_getThreadFactory(void);
void _deleteThreadFactory(void);

private:
```

```
ASC_Context(void){}

};

// This class provides a dummy message queue manager.
class _MsgQueueMgr : public MsgQueueMgr{

public:

_MsgQueueMgr(void){}
~_MsgQueueMgr(void){}

};
```

Both methods **_getThreadFactory** and **_deleteThreadFactory** must be implemented in the DCE library. **getMsgQueueMgr** and **deleteMsgQueueMgr** are implemented in the message queue manager.

Public methods

The following are the public methods.

threadMain

The **threadMain** method permanently blocks on the message queue waiting for requests. After it receives a request, it cleans up the remainder of a terminated thread.

If a thread object exits from **threadMain**, the mechanism discussed above is not used. Instead, the function invokes **threadMain** to do clean-up work.

Syntax:

```
TerminatorThread(char *ApplName):ASC_ThreadAppl(ApplName){} ~TerminatorThread(void){}
int threadMain(void **Rtn);
```

Inter-thread messaging system

The framework provides the inter-thread messaging system for the application to send messages. Although the design allows more than one type of message to be implemented, in this release only inter-thread messages are implemented.

The Inter-thread message allows the application to send messages among the threads. The implementation allows many threads to send and receive messages from the same queue. The mutex and condition are used to ensure that only one thread can access a message at a time.

This system supports both synchronous and asynchronous messages. When a thread sends an asynchronous message, it returns immediately. When a thread sends a synchronous message, it is blocked until the receiver frees it up. In addition, a sender or receiver can delete asynchronous messages, but only the sender can delete synchronous messages.

The **MsgQueueMgr** allows the application to create and retrieve message queues. After the application obtains the queue, an object of **ThreadMsgQueue**, it can use it to pass messages to other threads which monitor this queue, or retrieve messages from this queue. **Asc_MsgQueue** provides a generic interface which is redefined by its subclass such as **ThreadMsgQueue**.

ASC_Msg provides a default message object which is transferred among threads as vehicles to carry messages.

Because the system can have many message queues and each queue can contain many messages at a time, the system uses the Standard C++ containers to fulfill this.

ASC_Mutex is used to coordinate transferring the message among threads so that only one thread may access a message at a time.

Message queue manager class

MsgQueueMgr describes the message queue manager. It provides an interface for the application to obtain message queues. Because all message queues in a process are managed by the manager using the Standard C++ classes, this manager is not hit each time a thread accesses a queue. Instead, a thread keeps the queue locally after it obtains the queue. The object generated from this class is a singleton object.

Synopsis

```
class MsgQueueMgr {
public:

int getNumOfQueues(void);
ASC_MsgQueue* genThreadMsgQueue(char *QueueName,
int &QueueID);
ASC_MsgQueue* getMsgQueue(int &QueueID);
ASC_MsgQueue* getMsgQueue(char *QueueName);
void delMsgQueue(char *QueueName);
void delMsgQueue(int QueueID);
void delMsgQueue(ASC_MsgQueue *MsgQueue);
static MsgQueueMgr* getMsgQueueMgr(void);

protected:

~MsgQueueMgr(void){}
MsgQueueMgr(void){}

private:
    static MsgQueueMgr *m_TheQueueMgr;
ASC_Mutex m_TheMutex;
};
```

Public methods

The following are the public methods.

genThreadMsgQueue

```
ASC_MsgQueue* genThreadMsgQueue(char *QueueName, int &QueueID);
```

Use this function to generate a new message queue. If the message queue exists, this function returns the queue ID.

getMsgQueue

```
ASC_MsgQueue* getMsgQueue(int &QueueID);
ASC_MsgQueue* getMsgQueue(char *QueueName);
```

These functions retrieve a message queue.

delMsgQueue

```
void delMsgQueue(char *QueueName);
void delMsgQueue(int QueueID);
void delMsgQueue(ASC_MsgQueue *MsgQueue);
```

These functions remove a message queue.

getMsgQueueMgr

```
static MsgQueueMgr* getMsgQueueMgr(void);
```

Gets a pointer to the message queue manager. This function makes the message queue manager singleton.

Message queue class

MsgQueueMgr uses the Standard C++ container to manage the queue objects. The class is an abstract class that provides the application a generic interface to access messages.

Synopsis

```
class ASC_MsgQueue {
public:

    friend class MsgQueueMgr;
    ASC_MsgQueue(const char *MsgQueueName, const int MsgQueueID,
const int QueueType);
    ASC_MsgQueue(const char *MsgQueueName, const int MsgQueueType);
    ASC_MsgQueue(const int MsgQueueID, const int MsgQueueType);
    ASC_MsgQueue(const int MsgQueueType);

    virtual ~ASC_MsgQueue(void);
    void setQueueID(const int MsgQueueID) { m_QueueID = MsgQueueID; }
    void getQueueName(char *Name) const { strcpy (Name, m_QueueName); }
    const char *getQueueName(void) const { return m_QueueName; }
    void getQueueID(int &QueueID) const { QueueID = m_QueueID; }
    const int getQueueID(void) const { return m_QueueID; }
    const int getQueueType(void) const { return m_QueueType; }
    void addOneUser(void) { ++m_numOfUsers; }
    void removeOneUser(void) { --m_numOfUsers; }
    int getNumOfUsers(void) { return m_numOfUsers; }
    virtual int getQueueSize(void) const = 0;
    virtual int putMsg(ASC_Msg *Msg, const int SynchMsg) = 0;
    virtual int getMsg(ASC_Msg **Msg, const float WaitTime) = 0;
    virtual int getMsg(const int MsgType,
    ASC_Msg **Msg, const float WaitTime) = 0;
    virtual int peepMsg(ASC_Msg **Msg) = 0;
    virtual int peepMsg(const int MsgType, ASC_Msg **Msg) = 0;
    virtual void commitMsg(ASC_Msg *Msg) = 0;

protected:
private:
    ASC_MsgQueue(void):m_QueueName(0),m_QueueID(0),m_QueueType(0) {}
    char *m_QueueName;
    int m_QueueType;
```

```
    int m_QueueID;
    int m_numOfUsers;
    ASC_MsgQueue(const ASC_MsgQueue & TheQueue);
    ASC_MsgQueue & operator=(const ASC_MsgQueue & TheQueue);
};
```

Constructors

This constructor is used to construct a queue.

Syntax:

```
ASC_MsgQueue(const char *MsgQueueName, const int MsgQueueID, const int
QueueType);
```

These constructors are used to retrieve a message queue.

Syntax:

```
ASC_MsgQueue(const char *MsgQueueName, const int MsgQueueType);
ASC_MsgQueue(const int MsgQueueID, const int MsgQueueType);
ASC_MsgQueue(const int MsgQueueType);
```

Public methods

The following ar e the public methods.

addOneUser, removeOneUser, getNumOfUsers

These methods are used to keep track number of users are currently using a queue. This information is used for deleting a queue, because a queue can be deleted only when it is currently used by one user.

Syntax:

```
void addOneUser(void) { ++m_numOfUsers; }
void removeOneUser(void) { --m_numOfUsers; }
int getNumOfUsers(void) { return m_numOfUsers;//
```

getQueueSize

Gets how many messages are in a queue.

Syntax:

```
virtual int getQueueSize(void) const = 0;
```

putMsg

This member function inserts a message into the queue.

Syntax:

```
virtual int putMsg(ASC_Msg *Msg, const int SynchMsg) = 0;
```

getMsg

This member function retrieves a message from the top of the queue (the message staying in the queue the longest). Once the action is completed, the message is removed from the queue.

Syntax:

```
virtual int getMsg(ASC_Msg **Msg, const float WaitTime)= 0;
```

This member function retrieves the first message which matches the type (the message with the type staying in the queue the longest). Once the action is completed, the message is removed from the queue.

Syntax:

```
virtual int getMsg(const int MsgType, ASC_Msg **Msg, const float WaitTime) = 0;
```

peepMsg

These functions do not remove messages from a queue.

Syntax:

```
virtual int peepMsg(ASC_Msg **Msg)=0;
virtual int peepMsg(const int MsgType, ASC_Msg **Msg) = 0;
```

commitMsg

This function is used to commit a synchronous message after the receiver has received the message. Before the receiver calls this function, the sender thread is blocked.

Syntax:

```
virtual void commitMsg(ASC_Msg *Msg) = 0;
```

ThreadMsgQueue class

This class is generated by subclassing the abstract **ASC_MsgQueue** class to provide message queues for inter-thread communication. It essentially redefines all message queue interface functions.

Synopsis

```
class ThreadMsgQueue : public ASC_MsgQueue {
public:

    ThreadMsgQueue(const char *MsgQueueName, const int MsgQueueID,
    const int MsgQueueType = ASC_ThreadQueue);
    ThreadMsgQueue(const char *MsgQueueName,
    const int MsgQueueType = ASC_ThreadQueue);
    ThreadMsgQueue(const int MsgQueueID,
    const int MsgQueueType = ASC_ThreadQueue);
    ~ThreadMsgQueue(void);

    int getQueueSize(void) const { return m_MsgList.size(); }
    int putMsg(ASC_Msg *Msg, const int AsychMsg);
```

```
    int getMsg(ASC_Msg **Msg, const float WaitTime);
    int getMsg(const int MsgType,
    ASC_Msg **Msg, const float WaitTime);
    int peepMsg(ASC_Msg **Msg);
    int peepMsg(const int MsgType, ASC_Msg **Msg);
    void commitMsg(ASC_Msg *Msg) { Msg->commitMsgWait(); }
    inline std::vector<ASC_Msg*>::iterator
    find_by_type(std::vector<ASC_Msg*> &container, int type);

private:
    std::vector<ASC_Msg*> m_MsgList;
    ASC_Mutex *m_TheMutex;
    ASC_Condition *m_TheCond;
    ThreadMsgQueue(const ThreadMsgQueue & TheQueue);
    ThreadMsgQueue & operator=(const ThreadMsgQueue & TheQueue);
};
```

Public methods

The following are the public methods.

getQueueSize

Retrieves the number of messages in the queue.

Syntax:

```
int getQueueSize(void)const{return m_MsgList.size(); }
```

putMsg

Saves a message into the queue. This function returns immediately when it sends an asynchronous message.

Syntax:

```
int putMsg(ASC_Msg *Msg, const int AsychMsg);
```

getMsg

Retrieves a message. This function removes a message from the head of the queue.

Syntax:

```
int getMsg(ASC_Msg **Msg, const float WaitTime);
```

getMsg

Retrieves a message. This function removes a message from the queue which matches up with the type specified.

Syntax:

```
int getMsg(const int MsgType, ASC_Msg **Msg, const float WaitTime);
```

peepMsg

The following two functions are similar to get functions, however, they do not remove messages from the queue.

Syntax:

```
int peepMsg(ASC_Msg **Msg);
int peepMsg(const int MsgType, ASC_Msg **Msg);
```

commitMsg

Required to store the message.

Syntax:

```
void commitMsg(ASC_Msg *Msg) { Msg->commitMsgWait(); }
```

Message class

Use **ThreadMsg** to generate message objects. Message objects are transferred among threads. You can use this basic class to transfer messages containing the message type and message text. You can generate a customized message object through subclassing this class, to add desired message contents.

ThreadMsg also provides the means for the message queue object to handle synchronous and asynchronous messages.

Synopsis

```
class ASC_Msg {
public:

    ASC_Msg(int MsgType, void *Msg);
    ASC_Msg(int MsgType);
    virtual ~ASC_Msg(void);
    void setType(const int MsgType) { m_Type = MsgType; }
    void setMsg(void *Msg) { m_Text = Msg; }
    int getType(void) const { return m_Type; }
    void getType(int &MsgType) const { MsgType = m_Type; }
    void* getMsg(void) const { return m_Text; }
    void getMsg(void **Msg) const { *Msg = m_Text; }
    operator char*(void) { return (char *)m_Text; }
    operator int&(void) { return m_Type; }
    virtual void commitMsgWait(void);
    virtual void doMsgWait(void);
    void initSyn(void);

protected:
private:
    int m_Type;
    void *m_Text;
    int m_MsgStatus;
    ASC_Mutex *m_TheMutex;
    ASC_Condition *m_TheCond;
    ASC_Msg(void);
};
```

Constructors

```
ASC_Msg(int MsgType, void *Msg);
```

Creates a message.

Public methods

The following are the public methods.

doMsgWait

```
void doMsgWait(void)
```

This function blocks the sender and waits for the receiver to commit the message. This function is used only by **ASC_MsgQueue**.void commitMsgWait(void).

commitMsgWait

```
void commitMsgWait(void)
```

This function commits the message and releases the sender. The application can use either this one or one in **ASC_MsgQueue** to commit a message.

initSyn

```
void initSyn(void);
```

This function initializes a message.

XML JMX interface

For overview information on the JMX interface, refer to the *ASAP Server Configuration Guide*. For complete reference information on the JMX interface, refer to the *ASAP Online Reference*.

ASAP daemon API

The ASAP daemon consists of two parts: a daemon server and a group of client APIs for WebLogic Server application developers.

The ASAP daemon server is packaged in asaplibcommon.jar. It can be started without ASAP or WLS being activated and keeps running until the user stops the server by means of a script. For more information, refer to the *ASAP System Administrator's Guide*. The server always listens to requests issued from WLS applications. When the ASAP daemon receives a request, it performs the relevant I/O operation against the current ASAP instance.

The ASAP daemon client is not an independent program. It consists of the client APIs, plus the client configuration data in asap.ear. In asap.ear, the ASAP daemon client requires the following configuration arrangement (particularly those that appear in bold):

Contents of asap.ear: after compilation

```
<root>

ssam<ENV_ID>.jar
srp.jar
jsrp_connector.rar
jmx_connector.rar
sadtConsole.war
sadt.jar
<lib>
xercesImpl.jar
asapd_utils.jar
<META-INF>
weblogic-application.xml
application.xml
MANIFEST.MF
```

Java Program Sources:

- $ASAP_BASE/jshared/src – build.xml creates $ASAP_BASE/lib/asap.ear with the proper deployment descriptors, **jmx_connector.rar, asapd.jar**, jsrp_connector.rar and xercesImpl.jar
- $ASAP_BASE/jmx/src/web – build.xml adds sadtConsole.war to asap.ear
- $ASAP_BASE/jmx/src/ejb – build.xml adds sadt.jar to asap.ear
- $ASAP_BASE/jsrp/src – build.xml adds srp.jar to asap.ear.
- $ASAP_BASE/security/src – build.xml adds ssam.jar to asap.ear

The following configuration data must be customized before asap.ear is deployed in WebLogic Server.

Daemon server host and port, defined in META-INF/ra.xml within jmx_connector.rar:

- replace "__HOSTNAME__" and "__PORTNUMBER__" with real host and port values

Daemon client authorized user name, defined in WEB-INF/weblogic.xml within sadtConsole.war:

- replace "_USERNAME_" with real WLS login name

Daemon client used information, defined in WEB-INF/web.xml within sadtConsole.war:

- replace "ASAPDEV_ASAP_BASE" with ASAP_BASE directory (mirror) in WLS
- replace "ASAPDEV_SYBASE" with Sybase interfaces file path (mirror) in WLS
- replace "ASAPDEV_CTRL_PSWD" with current ASAP control password

> **Note:**
>
> Because ASAP_BASE and SYBASE file are mirrored on the WLS side, the daemon server can use actual paths.

The two mirrors can be relative paths, e.g., "A46". A relative path is based on the WLS domain directory. Never replace these paths with the real $ASAP_BASE path or the real $SYBASE/interfaces path if WLS is on the same machine as ASAP.

The Sybase interfaces file name in the WLS mirror should be same as on the ASAP side.

Replace "SAAS-1" with $ENV_ID in the WLS application JNDI prefix string in the following files:

- META-INF/weblogic-ra.xml within jmx_connector.rar
- WEB-INF/web.xml within sadtConsole.war
- WEB-INF/weblogic.xml within sadtConsole.war
- META-INF/weblogic-ejb-jar.xml with sadt.jar

The necessary "Class-Path" information will be properly defined in the related MANIFEST.MF files.

Daemon client APIs

The asapd.jar provides daemon client APIs to WebLogic Server application developers to provide a consistent API for the internal protocols of the daemon process.

The ASAP daemon client APIs are based on two primary classes: RemoteFile, RemoteCommand. They have a common superclass RemoteAccess, which defines some common constant data and methods.

RemoteFile

A remote file can be either a file or a directory in remote site, that is, a daemon server site. The methods of this class are designed to fully or partially substitute for the same/similar methods in Java File class.

Syntax:

```
public RemoteFile( ASAPDataSource _ds, String remotePath, int pathBase)
throws Exception
```

Constructors:

- **_ds:** An object of ASAPDataSource (this class is provided in asapd_utils.jar); a factory of connections to the daemon server process.
- **remotePath:** A string of path of a remote file/directory that the client wants to access.
- **pathBase:** An integer that represents the remote path base. The supported bases are:
  - **RemoteFile.ROOT_BASED** – the remotePath is an absolute path
  - **RemoteFile.ASAP_BASED** – the remotePath is a path relative to the remote $ASAP_BASE
  - **RemoteFile.SYBASE_BASED** – the remotePath is a path relative to the remote $SYBASE

Property checking methods

```
public boolean canRead()
public boolean canWrite()
public boolean exists()
public boolean isFile()
public boolean isDirectory()
public boolean isHidden()
```

```
public boolean isAbsolute()
public long lastModified()
public long length()
public String toString()
public String getPath()
public String getName()
public String getParent()
```

All these methods behave in the same way as Java's File class, except for the following conditions.

- if a RemoteFile isDirectory, calling length() is not relevant.
- if a RemoteFile does not exist, all other state checking return false.
- all the properties have their values same as the RemoteFile constructor was called.

get Methods

This method provides WLS client with the ability to retrieve a file from a remote site. The file path has been specified in the constructor.

Syntax:

```
public InputStream get()
throws Exception
```

put Methods

These methods provide the WLS client with the ability to save a set of data to a remote file. The file path has been specified in the constructor.

- **put(byte[] buffer**) is used primarily to save any data set in memory to a remote file.
- **put(InputStream is, long size)** can be called after the “get" of an InputStream, which is usually of a subclass of InputStream, such as FileInputStream or PipedInputStream.
- **put(String localFilePath)** is used to transfer a local file to a remote file whose path has been specified in the constructor. This method is designed to minimize modification to the existing WLS application. The application can get a remote file, edit it, and then save it locally (the WLS application writes files back in many different ways depending on the particular logic).

Syntax:

```
public void put(byte[] buffer)
throws Exception
public void put(InputStream is, long size)
throws Exception
public void put(String localFilePath)
throws Exception
```

RemoteCommand

A remote command can be either a special daemon command or a UNIX script in a remote site, that is, a daemon server site.

Syntax:

```
public RemoteCommand(ASAPDataSource _ds, CommandInfo _commInfo)
throws Exception
```

Constructors:

- **_ds:** An object of ASAPDataSource (this class is provided in jmx_connector.jar); a factory of connection to the daemon server process.
- **_commInfo:** A CommandInfo that represents the command property, including:
  - public byte command - the command, which can be REM_REQ_COPY_FILE – copy a file from source file/dir to dest file/dir; REM_REQ_MOVE_FILE – move a file from source file/dir to dest file/dir; REM_REQ_EXTRACT_JFILE – unjar a dest file from a source jar file
  - public byte base_dir - the path base, same as RemoteFile
  - public String src_dir - the resource directory
  - public String src_file - the resource file
  - public String dest_dir - the destination directory
  - public String dest_file - the destination file

action Methods

These three methods provide WLS client with the ability to actually perform the command specified in the constructor.

```
public void copy()

throws Exception

public void move()

throws Exception

public void unjar()

throws Exception
```

4
Provisioning Interfaces

This chapter describes the following provisioning (upstream) interfaces:

- SARM configuration interface
- SARM provisioning interface
- C++ SRP API library

SARM configuration interface

This section covers the functions for the SARM and includes the following subsections:

- Static table configuration
- Error management
- Switch blackout processing
- Switch direct interface (SWD)
- Stop work order interface
- Localizing International Messages

Static table configuration

To interface to the static configuration database tables, use the function-based interface instead of SQL insert scripts.

The function-based interface reduces the dependency between administrators who configure the system and product developers who need to make changes to the static tables to support new functionality.

This section lists the syntax, descriptions, parameters, and results for the SARM configurations and includes the delete, list, and new procedures interface definitions.

SSP_db_admin

This function performs customer-specific database administration functions, such as purging the database of completed orders, and removing orphaned definitions from the SARM database.

For more information on this function and database purging, refer to the *ASAP System Administrator's Guide*.

For more information about using functions, see "Oracle Execution Examples."

Affected tables:

- tbl_wrk_ord
- tbl_asap_stats

- tbl_info_parm
- tbl_srq
- tbl_srq_csdl
- tbl_srq_log
- tbl_asdl_log
- tbl_srq_parm
- tbl_srq_asdl_parm
- tbl_wo_event_queue
- tbl_wo_audit
- tbl_usr_wo_prop
- tbl_aux_wo_prop

Table 4-1 SSP_db_admin Parameters

Name	Description	Req'd	(I)nput/ (O)utput
days	Specifies the age (in days) of work orders to delete. All completed work orders older than the specified number of days are deleted.	Yes	I

SSP_gather_asap_stats

Gathers statistics for objects in the database (tbl_wrk_ord). The information includes the distribution of data, the number of rows in the table and other important statistics. Statistics gathering is governed by the following parameters in ASAP.cfg:

- GATHER_STATS
- GATHER_STATS_PROC
- DB_PCT_ANALYZE
- DB_PCT_ANALYZE_IDX
- GATHER_DEGREE
- DB_ADMIN_TIME

For more information about using functions, see "Oracle Execution Examples."

Syntax:

```
SSP_gather_asap_stats(
a_tab_estimate_pct number,
a_ind_estimate_pct number,
a_degree number)
```

Table 4-2 SSP_del_asdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
a_tab_estimate_pct number	This parameter applies to Oracle only. It is used to update statistics on all user-defined tables. The updates are done when the database administrations tasks are performed. (See also DB_ADMIN_TIME.) This parameter is used to optimize the database query performance. The Oracle SQL statement is “analyze table table_name estimate statistics sample DB_PCT_ANALYZE percent". For further information, refer to “Analyze Command" in the Oracle SQL Reference manual.	Yes	I
a_ind_estimate_pct number	Percentage of the index to analyze when gathering statistics.	Yes	I
a_degree number	Degree of parallelism when gathering statistics. The degree parameter can take the value of auto_degree. When you specify the auto_degree, Oracle determines the degree of parallelism automatically. It will be either 1 (serial execution) or default_degree (the system default value based on number of CPUs and initialization parameters), according to the size of the object.	Yes	I

SSP_del_asdl_defn

This function deletes ASDL definitions from tbl_asdl_config. Wildcards are permitted.

> **Note:**
>
> If you do not specify an ASDL command, all ASDL command definitions are deleted from the configuration records.

Syntax:

```
var retval number;
exec :retval :=  SSP_del_asdl_defn (['asdl_cmd'])
```

Example:

```
var rc refcursor;
var retval number;
exec :retval :=  SSP_del_asdl_defn (:rc,'M-CREATE_SINGLE_LINE_ACCESS');
```

This example removes the **M-CREATE_SINGLE_LINE_ACCESS** configuration record from the static configuration tables.

For more information about using functions, see "Oracle Execution Examples."

Table 4-3 SSP_del_asdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command to be deleted.	No	I

SSP_del_asdl_map

This function deletes ASDL-to-State Table mappings from tbl_nep_asdl_prog. The mapping is based on the technology and software load.

For more information about using functions, see "Oracle Execution Examples."

Syntax:

```
var retval number;
exec :retval :=  SSP_del_asdl_map ['tech'] [, 'sftwr_load'] [, 'asdl_cmd']
```

Table 4-4 SSP_del_asdl_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
tech	The technology type of NE or SRP with which the Interpreter is to interact.	No	I
sftwr_load	The version of the software currently running on the NEP or SRP.	No	I
asdl_cmd	The ASDL command.	No	I

SSP_del_asdl_parm

This function deletes an ASDL parameter from tbl_asdl_parm.

> **Note:**
>
> If you do not enter a sequence number, all parameters associated with this ASDL command are deleted.

For more information about using functions, see "Oracle Execution Examples."

This example removes the ASDL command parameter with the sequence number 4 from the configuration table for the command **M-CREATE_SINGLE_LINE_ACCESS**.

Table 4-5 SSP_del_asdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command.	No	I
parm_seq_no	The parameter sequence number.	No	I

SSP_del_clli_map

This function deletes a remote CLLI to host CLLI mapping in tbl_clli_route.

For more information about using functions, see "Oracle Execution Examples."

Table 4-6 SSP_del_clli_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
mach_clli	The remote NE.	No	I
asdl_cmd	ASDL command.	No	I

SSP_del_comm_param

This function deletes communication parameter information from tbl_comm_param.

For more information about using functions, see "Oracle Execution Examples."

Table 4-7 SSP_del_comm_param Parameters

Name	Description	Req'd	(I)nput/ (O)utput
dev_type	The device type. Choose the type of connection from the following: • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection • C – CORBA	No	I
host	Host CLLI. Set to COMMON_HOST_CFG or the host CLLI associated with the command processor. For a common host, the parameter value is the default value, otherwise, it is host-specific.	No	I
device	The physical or logical device name. Set to COMMON_DEVICE_CFG or the device associated with the command processor. For a common device, the parameter value is the default value, otherwise, it is device-specific.	No	I
param_label	Specifies the communication parameter label.	No	I
param_value	Specifies the communication parameter value.	No	I
param_desc	Specifies the communication parameter description.	No	I

SSP_del_csdl_asdl

This function deletes a CSDL-to-ASDL mapping definition from tbl_csdl_asdl.

> **Note:**
>
> If you do not specify an ASDL command sequence number, all mapping relationships for the specified CSDL command are removed from the configuration tables. If you do not specify a CSDL command, all mapping relationships are removed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-8 SSP_del_csdl_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_cmd	The CSDL command.	No	I
asdl_seq_no	The sequence number of the ASDL command.	No	I

SSP_del_csdl_defn

This function deletes CSDL definitions from tbl_csdl_config.

> **Note:**
>
> If you do not specify a CSDL command, all CSDL command definitions are removed.

Syntax:

```
var retval number;
exec :retval :=  SSP_del_csdl_defn ['csdl_cmd']
```

Example:

```
var retval number;
exec :retval :=  SSP_del_csdl_defn 'M-CREATE_BUS_LINE'
```

The configuration record for M-CREATE_BUS_LINE is removed from the configuration tables.

For more information about using functions, see "Oracle Execution Examples."

Table 4-9 SSP_del_csdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_cmd	The CSDL command.	No	I

SSP_del_dn_map

This function deletes a directory number mapping from tbl_nep_rte_asdl_nxx.

For more information about using functions, see "Oracle Execution Examples."

Table 4-10 SSP_del_dn_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command identifier.	No	I
npa	The Numbering Plan Area code.	No	I
nxx	The Central Office code.	No	I
from_line	The lowest line number in the range.	No	I
to_line	The highest line number in the range.	No	I

SSP_del_id_routing

This function deletes a host NE and the ID_ROUTING mapping record from tbl_id_routing. Use this function when routing by ID_ROUTING is used.

For more information about using functions, see "Oracle Execution Examples."

Table 4-11 SSP_del_id_routing Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier.	Yes	I
asdl_cmd	The ASDL command.	Yes	I
id_routing_from	The starting point of a range of ID_ROUTING.	Yes	I
id_routing_to	The end point of a range of ID_ROUTING.	Yes	I

SSP_del_intl_msg

This function deletes an international message record from the tbl_msg_convert.

If you do not specify a message identifier, all messages with the specified language code are deleted.

For more information about using functions, see "Oracle Execution Examples."

Table 4-12 SSP_del_intl_msg Parameters

Name	Description	Req'd	(I)nput/ (O)utput
lang_cd	The language code.	No	I
msg_id	The unique message identifier for the message to be removed from the SARM database.	No	I

The following example shows how to delete an international message:

```
var retval number;
exec :retval :=  SSP_del_intl_msg 'USA', 1
```

This deletes American English message 1 from the SARM database.

SSP_del_ne_host

This function deletes a host NE definition from tbl_host_clli.

For more information about using functions, see "Oracle Execution Examples."

Table 4-13 SSP_del_ne_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	No	I

SSP_del_nep

This function deletes an NEP secondary pool definition from tbl_nep.

For more information about using functions, see "Oracle Execution Examples."

Table 4-14 SSP_del_nep Parameters

Name	Description	Req'd	(I)nput/ (O)utput
nep_svr_cd	The NEP managing the secondary pool of devices.	No	I

SSP_del_nep_program

This function deletes State Table actions based on the specified program name and/or line number from tbl_nep_program.

If the line number is not supplied, all actions with positive line numbers are removed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-15 SSP_del_nep_program Parameters

Name	Description	Req'd	(I)nput/ (O)utput
program	The State Table program identifier.	Yes	I
line_no	The State Table line number to delete. If set to NULL, all lines of the State Table are deleted.	No	I

SSP_del_net_elem

This function deletes an NE definition for an NEP from the SARM database.

This function deletes a network elements from tbl_ne_config.

For more information about using functions, see "Oracle Execution Examples."

Table 4-16 SSP_del_net_elem Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	No	I
nep_svr_cd	The logical name of the NEP server that connects to the host NE.	No	I

SSP_del_resource

This function deletes a device from **tbl_resource_pool.**

For more information about using functions, see "Oracle Execution Examples."

Table 4-17 SSP_del_resource Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asap_sys	The ASAP environment (TEST, PROD, and so on).	No	I
device	The physical or logical device name.	No	I

SSP_del_srp

This function deletes an SRP definition from the tbl_asap_srp.

For more information about using functions, see "Oracle Execution Examples."

Table 4-18 SSP_del_srp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srp_id	The logical SRP name.	No	I

SSP_del_stat_text

This function deletes static text labels used in the OCA GUI client from tbl_stat_text.

For more information about using functions, see "Oracle Execution Examples."

Table 4-19 SSP_del_stat_text Parameters

Name	Description	Req'd	(I)nput/ (O)utput
stat_id	The logical group of static text messages such as WO_STATE. If this is not specified, all entries in tbl_stat_text are deleted.	No	I
status	The integer identifier for member of logical grouping.	No	I
code	The string identifier for member of logical grouping.	No	I

SSP_del_user_err_threshold

This function deletes a user-defined error threshold or set of thresholds from tbl_user_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-20 SSP_del_user_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	The ASDL command.	No	I
user_type	The user-defined error type.	No	I

SSP_del_userid

This function deletes a user ID from tbl_uid_pwd.

For more information about using functions, see "Oracle Execution Examples."

Table 4-21 SSP_del_userid Parameters

Name	Description	Req'd	(I)nput/ (O)utput
uid	The user ID.	No	I

SSP_get_async_ne

This function returns the names of all the NEs that have a ASYNC_CONN communication parameter defined with a value of TRUE or FALSE. The existence of a

parameter labeled ASYNC_CONN indicates that the NE has an asynchronous interface. The parameter value of TRUE or FALSE indicates whether the NEP server should establish an NE element connection at NEP start-up. This function has no input parameters.

For more information about using functions, see "Oracle Execution Examples."

Table 4-22 SSP_get_async_ne Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host	The host name of the NE having an ASYNCH_CONN communication parameter defined.	No	O
param_value	Value of the ASYNCH_CONN parameter - either TRUE or FALSE. Indicates whether the NEP server should establish an NE element connection at NEP start-up.	No	O

SSP_get_user_routing

This function returns a Host NE (host_clli) that is used to route the ASDL from tbl_user_routing. You must write this function and the associated database table based on the pre-defined interfaces and your own routing algorithm when using a user-defined routing.

For more information about using functions, see "Oracle Execution Examples."

Table 4-23 SSP_get_user_routing Parameters

Name	Description	Req'd	(I)nput/ (O)utput
user_routing	The USER_ROUTING defined as a work order parameter.	No	I
asdl_cmd	The ASDL command.	No	I
host_clli	The host NE identifier to be routed. If it fails to find the host NE associated with the USER_ROUTING, it returns NULL.	No	O

SSP_list_asdl

This function retrieves ASDL-related information from tbl_asdl_config, tbl_asdl_parm, tbl_nep_asdl_prog.

For more information about using functions, see "Oracle Execution Examples."

Table 4-24 SSP_list_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
RC2	Oracle Database Ref Cursor.	Yes	I/O
RC3	Oracle Database Ref Cursor.	Yes	I/O

Table 4-24 (Cont.) SSP_list_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl	The name of the ASDL.	No	I

The shaded groupings below indicate that multiple result sets are returned: in this case, three sets of data.

For more information about using functions, see "Oracle Execution Examples."

Table 4-25 SSP_list_asdl Results

Name	Datatype	Description
reverse_asdl	TYP_asdl_cmd	reverse ASDL command
ignore_rollback	TYP_yes_no	ignore rollback flag
route_flag	TYP_route	ASDL routing flag
description	varchar(40)	ASDL command description
asdl_lbl	TYP_parm_lbl	ASDL parameter label
csdl_lbl	TYP_parm_lbl	CSDL parameter label
param_typ	TYP_parm_typ	ASDL parameter type
default_vlu	TYP_parm_vlu	ASDL parameter default value
tech	TYP_tech	technology
sftwr_load	TYP_load	software loads
program	TYP_program	State Table program

SSP_list_asdl_defn

This function lists all or specific ASDL definitions from the SARM database (tbl_asdl_config, tbl_csdl_asdl). You can use wildcards in this procedure. If you do not specify a parameter, all ASDL definitions are returned.

Syntax:

```
var rc refcursor;
var retval number;
exec :retval :=  SSP_list_asdl_defn (:rc[,'asdl_cmd']);
print rc;
```

Example:

```
var rc refcursor;
var retval number;
exec :retval := SSP_list_asdl_defn (:rc,'M-CREATE_SINGLE_LINE_ACCESS');
print rc;
```

The following example lists the information for M-SINGLE_LINE_ACCESS:

```
asdl_cmd                    reverse_asdl                ignore_rollback
route_flag description
-------- ---------- ---------------------------------------
```

```
M-CREATE_SINGLE_LINE_ACCESS  M-DELETE_SINGLE_LINE_ACCESS    N              B
Create a single-line access service.
(1 row affected, return status = 0)
```

For more information about using functions, see "Oracle Execution Examples."

Table 4-26 SSP_list_asdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asdl_cmd	The ASDL command identifier.	No	I

Table 4-27 SSP_list_asdl_defn Results

Name	Datatype	Description
asdl_cmd	TYP_asdl_cmd	ASDL command.
reverse_asdl	TYP_asdl_cmd	Reverse ASDL command.
ignore_rollback	TYP_yes_no	Ignore rollback flag.
route_flag	TYP_route	ASDL routing flag.
description	varchar(40)	ASDL command description.

SSP_list_asdl_map

This function lists ASDL-to-State Table mappings according to various criteria. All parameters take wildcards. This function retrieves asdl mapping information from tbl_nep_asdl_prog.

Syntax:

```
var rc1 refcurson;
var retval number;
exec :retval :=  SSP_list_asdl_map (:RC1) [, 'tech'] [, 'sftwr_load'] [, 'asdl_cmd']
[, 'interpreter_type']
```

For more information about using functions, see "Oracle Execution Examples."

Table 4-28 SSP_list_asdl_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
tech	The technology type of the NE or SRP with which the Interpreter is to interact.	No	I
sftwr_load	The version of the software currently running on the NEP or SRP.	No	I
asdl_cmd	The ASDL command. Wildcards are accepted.	No	I
program	The State Table program.	No	I
interpreter_type	A value of 'S' indicates a State Table interpreter. A value of **J** indicates a JInterpreter. A null value defaults to **S**.	No	I

Table 4-29 SSP_list_asdl_map Results

Name	Datatype	Description
tech	TYP_tech	Technology.
sftwr_load	TYP_load	Software loads.
asdl_cmd	TYP_asdl_cmd	ASDL command.
program	TYP_program	State Table program.

SSP_list_asdl_parm

This function lists ASDL parameters from the SARM database (tbl_asdl_parm) by ASDL command name and/or ASDL parameter label. Wildcards are allowed.

Syntax:

```
var rc1 refcurson;
var retval number;
exec :retval :=  SSP_list_asdl_parm (:RC1) [, 'asdl_cmd'] [, 'asdl_parm_lbl']
```

Example:

```
var rc refcurson;
var retval number;
exec :retval :=  SSP_list_asdl_parm 'M-CREATE_SINGLE_LINE_ACCESS', 'NPA'
```

This example retrieves configuration information for the NPA parameter for the ASDL command **M-CREATE_SINGLE_LINE_ACCESS**.

For more information about using functions, see "Oracle Execution Examples."

Table 4-30 SSP_list_asdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asdl_cmd	The ASDL command.	No	I
asdl_parm_lbl	The ASDL parameter label.	No	I

Table 4-31 SSP_list_asdl_parm Results

Name	Datatype	Description
asdl_cmd	TYP_asdl_cmd	ASDL command.
parm_seq_no	TYP_seq_no	ASDL parameter number.
asdl_lbl	TYP_parm_lbl	ASDL parameter label.
csdl_lbl	TYP_parm_lbl	CSDL parameter label.
default_vlu	TYP_parm_vlu	ASDL parameter default value.
param_typ	TYP_parm_typ	ASDL parameter type.

SSP_list_clli_map

This function lists remote CLLI-to-Host CLLI mapping definitions that are contained in tbl_clli_route. All parameters take wildcards.

For more information about using functions, see "Oracle Execution Examples."

Table 4-32 SSP_list_clli_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
mach_clli	The remote NE.	No	I
host_clli	The host NE identifier of an NE or SRP.	No	I

Table 4-33 SSP_list_clli_map Results

Name	Datatype	Description
mach_clli	TYP_clli	Remote NE.
host_clli	TYP_clli	Host NE.
asdl_cmd	TYP_asdl_cmd	ASDL command identifier.

SSP_list_comm_param

This function lists communication parameters based on dev_type, host, device, param_label, or for all of them. This information is retrieved from tbl_comm_param.

For more information about using functions, see "Oracle Execution Examples".

Table 4-34 SSP_list_comm_param Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
dev_type	The device type. Choose from the following types of connections: • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection • C – CORBA	No	I

Table 4-34 (Cont.) SSP_list_comm_param Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host	The host CLLI. It is set to COMMON_HOST_CFG or the host CLLI associated with the command processor. If a common host, the parameter value is the default value, otherwise, it is host-specific.	No	I
device	The physical or logical device (port) name. Set to COMMON_DEVICE_CFG or the device associated with the command processor. If a common device, the parameter value is the default value, otherwise, it is device-specific.	No	I
param_label	The communication parameter label.	No	I

Table 4-35 SSP_list_comm_param Results

Name	Datatype	Description
dev_type	TYP_dev_type	Type of connection. Choose from: • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection
host	TYP_clli	Host CLLI. It is set to COMMON_HOST_CFG or the host CLLI associated with the command processor. If a common host, the parameter value is the default value, otherwise, it is host-specific.
device	TYP_device	The physical or logical device name. Set to COMMON_DEVICE_CFG or the device associated with the command processor. If a common device, the parameter value is the default value, otherwise, it is device-specific.
param_label	TYP_parm_lbl	Specifies the communication parameter label.
param_value	TYP_perf_parm_vlu	Specifies the communication parameter value.
param_desc	TYP_parm_desc	Specifies the communication parameter description.

SSP_list_csdl

This function retrieves CSDL-related information from tables tbl_csdl_config and tbl_csdl_asdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-36 SSP_list_csdl Parameters

Name	Description	Req'd	(I)nput/(O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
RC2	Oracle Database Ref Cursor.	Yes	I/O
csdl	The name of the CSDL.	Yes	I

Table 4-37 SSP_list_csdl Results

Name	Datatype	Description
asdl_cmd	TYP_asdl_cmd	ASDL command.
cond_flag	TYP_cond_flag	Condition flag.
label	TYP_parm_lbl	Parameter label to test the condition flag.
value	TYP_parm_vlu	Parameter value associated with the label.
rollback_req	TYP_yes_no	Flag indicating whether rollback is required.
csdl_level	TYP_csdl_level	The level of the CSDL in the SRQ. An integer between 0 and 255 that indicates the sequence level for the CSDL command within the work order. The SARM uses this integer to determine the order in which to provision CSDL commands from an SRP and then provisions CSDL commands that have lower level numbers first. Sequence levels are only relevant for inter-dependent CSDL commands.
fail_event	TYP_code	The system event to be triggered upon CSDL failure.
complete event	TYP_code	The system event to be triggered upon CSDL completion.
option_asdl	TYP_seq_no	Not used.
description	varchar(40)	ASDL command description.

SSP_list_csdl_asdl

This function lists CSDL-to-ASDL mapping definitions contained in tbl_csdl_asdl. Wildcards are allowed.

Syntax:

```
var rc refcursor;
var retval number;
exec :retval :=  SSP_list_csdl_asdl (:rc [,'csdl_cmd'] [, 'base_seq_no'] [,'asdl_cmd']
[,'cond_flag'] [,'parm_lbl'] [,'parm_vlu'] [,'eval_exp'])
```

Example:

```
var rc refcursor;
var retval number;
exec :retval :=  SSP_list_csdl_asdl (:rc,'M-CREATE_BUS_LINE');
```

This example lists all mapping relationships associated with the CSDL command M-CREATE_BUS_LINE as follows:

```
csdl_cmd                    asdl_seq_no asdl_cmd
cond_flag label
value



---------- ------------------------------ --------- --------
M-CREATE_BUS_LINE                     1
M-CLEAR_INTERCEPT               A
M-CREATE_BUS_LINE                     2
M-CREATE_SINGLE_LINE_ACCESS     A
M-CREATE_BUS_LINE                     3 ADD_ALWAYS_ON_3WC               D
ALWAYS_ON_AREA
M-CREATE_BUS_LINE                     4 ADD_ALWAYS_ON_CRT               D
ALWAYS_ON_AREA

(4 rows affected, return status = 0)
```

Table 4-38 SSP_list_csdl_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
csdl_cmd	The CSDL command.	No	I
asdl_cmd	The ASDL command.	No	I
cond_flag	Used to specify conditions that need to be met in order for the SARM to generate the ASDL command for the CSDL command. Type one of the following values: • **A** – Always generates the ASDL command for the CSDL command • **D** – Generates the ASDL command if the CSDL parameter is defined (present) • **N** – Generates the ASDL command if the CSDL parameter is not defined (present) • **E** – Generates the ASDL command if the CSDL parameter is defined and equal to a value. The generation of each ASDL command depends upon the results of the previous ASDL. When the previous command completes successfully, it returns parameters to the SARM. When using 'cond_flag'='E', the following values are required: • 'lbl1' • 'lbl2' • 'val1' • 'val2' When using 'cond_flag'='D' or 'N', the following values are required: • 'lbl1' • 'lbl2'	No	I

Table 4-38 (Cont.) SSP_list_csdl_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
parm_lbl and parm_vlu parameters	Required when you use CSDL parameter-dependent conditions. Set the CSDL command parameter name for 'D', 'N', and 'E' condition flags using **parm_lbl**. The 'E' condition flag checks that the CSDL command parameter is equal to the value specified by **parm_vlu**. For more information about these condition flags, refer to the previous parameter, **cond_flag**.	No	I
eval_exp	Contains combination of parameter names, operators, and values to which the parameters are compared.	No	I

Table 4-39 SSP_list_csdl_asdl Results

Name	Datatype	Description
csdl_cmd	TYP_csdl_cmd	CSDL command name.
asdl_seq_no	TYP_seq_no	ASDL command sequence number.
asdl_cmd	TYP_asdl_cmd	ASDL command.
cond_flag	TYP_cond_flag	Specifies conditions that need to be met for the SARM to generate the ASDL command for the CSDL command. One of the following values: • **A** – Always generates the ASDL command for the CSDL command • **D** – Generates the ASDL command if the CSDL parameter is defined (present) • **N** – Generates the ASDL command if the CSDL parameter is not defined (present) • **E** – Generates the ASDL command if the CSDL parameter is defined and equal to a value. Each ASDL command generation depends upon the results of previous ASDL commands on the work order which returned parameters to the SARM upon successful ASDL command completion. When using 'cond_flag'='E', the following values are required: • 'lbl1' • 'lbl2' • 'val1' • 'val2' When using 'cond_flag'='D' or 'N', the following values are required: • 'lbl1' • 'lbl2'
label	TYP_parm_lbl	Parameter label to test the condition flag.

Table 4-39 (Cont.) SSP_list_csdl_asdl Results

Name	Datatype	Description
value	TYP_parm_vlu	Parameter value associated with the label.

SSP_list_csdl_defn

This function lists configuration information for the CSDL command you specify from the SARM database (tbl_csdl_config). This information includes the rollback flag, CSDL command sequence number, fail and completion events, and a description of the command. If you do not specify a CSDL command, the procedure returns information on all CSDL commands currently defined in the SARM.

Syntax:

exec :retval := SSP_list_csdl_defn (:RC1[, 'csdl_cmd']

Example:

```
var rc refcusror;
var retval number;
exec :retval :=  SSP_list_csdl_defn (:rc,'M-CREATE_BUS_LINE')
```

For more information about using functions, see "Oracle Execution Examples."

Table 4-40 SSP_list_csdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
csdl_cmd	The CSDL command identifier.	No	I

Table 4-41 SSP_list_csdl_defn Results

Name	Datatype	Description
csdl_cmd	TYP_csdl_cmd	CSDL command name.
rollback_req	TYP_yes_no	Flag indicating whether rollback is required.
csdl_level	TYP_csdl_level	The level of the CSDL in the SRQ. An integer between 0 and 255 that indicates the sequence level for the CSDL command within the work order. The SARM uses this integer to determine the order in which to provision CSDL commands from an SRP and then provisions CSDL commands that have lower level numbers first. Sequence levels are only relevant for inter-dependent CSDL commands.
fail_event	TYP_code	The system event to be triggered upon CSDL failure.
complete_event	TYP_code	The system event to be triggered upon CSDL completion.
description	varchar(40)	ASDL command description.

SSP_list_dn_map

This function lists directory mappings for ASDL command, directory, exchange number, or for all of them from tbl_nep_rte_asdl_nxx.

For more information about using functions, see "Oracle Execution Examples."

Table 4-42 SSP_list_dn_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asdl_cmd	The ASDL command.	No	I
npa	The Numbering Plan Area code.	No	I
nxx	The Central Office code.	No	I

Table 4-43 SSP_list_dn_map Results

Name	Datatype	Description
asdl_cmd	TYP_asdl_cmd	ASDL command identifier.
npa	TYP_npa	The Numbering Plan Area code.
nxx	TYP_nxx	The Central Office code.
from_line	TYP_line	Beginning LINE of DN range.
to_line	TYP_line	End LINE of DN range.
queue_nm	TYP_clli	Host NE to which the DN routing applies.

SSP_list_host

This function retrieves host-related information from the following tables: tbl_resource_pool, tbl_ne_config, tbl_clli_route.

For more information about using functions, see "Oracle Execution Examples."

Table 4-44 SSP_list_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
RC2	Oracle Database Ref Cursor.	Yes	I/O
host	The host CLLI identifier.	Yes	I

Table 4-45 SSP_list_host Results

Name	Datatype	Description
device	TYP_device	The name of the logical device to be used to establish a connection to the NE.

Table 4-45 (Cont.) SSP_list_host Results

Name	Datatype	Description
line_type	TYP_dev_type	The communication protocol used by the specified device.
vs_key	TYP_long	Reserved. The shared memory segment identifier for the Virtual Screen buffer.
mach_clli	TYP_clli	The remote NE identifier.
asdl_cmd	TYP_asdl_cmd	The ASDL command.

SSP_list_id_routing

This function lists the host NE and the ID_ROUTING mapping record in tbl_id_routing. You can use this function when routing by ID_ROUTING is used.

For more information about using functions, see "Oracle Execution Examples."

Table 4-46 SSP_list_id_routing Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	No	I/O
host_clli	The host NE identifier.	Yes	I

Table 4-47 SSP_list_id_routing Results

Name	Datatype	Description
host_clli	TYP_clli	The host NE identifier.
asdl_cmd	TYP_asdl_cmd	The ASDL command.
id_routing_from	Varchar	The starting point of a range of ID_ROUTING.
id_routing_to	Varchar	The end point of a range of ID_ROUTING.

SSP_list_intl_msg

This function lists the international message records from tbl_msg_convert. It either lists all messages for a specified language or, if you specify a message identifier, a single record.

A list of core ASAP messages is contained in the *ASAP System Administrator's Guide*.

For more information about using functions, see "Oracle Execution Examples."

Table 4-48 SSP_list_intl_msg Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O

Table 4-48 (Cont.) SSP_list_intl_msg Parameters

Name	Description	Req'd	(I)nput/ (O)utput
lang_cd	The language code.	No	I
msg_id	Unique message identifier.	No	I

Table 4-49 SSP_list_intl_msg Results

Name	Datatype	Description
lang_cd	TYP_lang_cd	The language code.
msg_id	TYP_unid	Message identifier.
msg_type	TYP_msg_typ	Message formatting types: • D – Dynamic • S – Static
message	varchar(255)	Message text.
var_description	varchar(40)	Description of the substitute fields.
wo_audit	TYP_wo_audit	Destination for the log message.

SSP_list_ne_host

This function lists host NE definitions from tbl_host_clli. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-50 SSP_list_ne_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
host_clli	The host NE identifier of an NE or SRP.	No	I

Table 4-51 SSP_list_ne_host Results

Name	Datatype	Description
host_clli	TYP_clli	The host NE identifier of an NE or SRP.
tech_type	TYP_tech	Technology type.
sftwr_load	TYP_load	Software loads.

SSP_list_nep

This function lists NEP secondary pool definitions, stored in tbl_nep. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-52 SSP_list_nep Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
nep_svr_cd	The NEP managing the secondary pool of devices.	No	I

Table 4-53 SSP_list_nep Results

Name	Datatype	Description
nep_svr_cd	TYP_code	The NEP managing the secondary pool of devices.
dialup_pool	TYP_pool	Secondary pool of devices.

SSP_list_nep_program

This function lists the State Table program to be used within the Interpreter. This information is retrieved from tbl_nep_program.

For more information about using functions, see "Oracle Execution Examples."

Table 4-54 SSP_list_nep_program Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
program	The State Table program name.	No	I

Table 4-55 SSP_list_nep_program Results

Name	Datatype	Description
program	TYP_program	The name of the State Table program.
line_no	int	The line number in the State Table program.
action	TYP_action	The action string used to identify the action performed by the Interpreter in the command processor.
act_string	TYP_action_string	The action string associated with the State Table action.
act_int	int	The action integer which represents the next line number in the State Table program at which the execution of the State Table should continue, or a numeric field specific to the particular action function.

SSP_list_net_elem

This function lists NE definitions based on the host NE and/or NEP server you specify. This function lists NE definitions from tbl_ne_config. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-56 SSP_list_net_elem Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
host_clli	The host NE identifier of an NE or SRP.	No	I
nep_svr_cd	The logical name of the NEP server that connects to this host NE.	No	I

Table 4-57 SSP_list_net_elem Results

Name	Datatype	Description
host_clli	TYP_clli	The host NE.
nep_svr_cd	TYP_code	The logical name of the NEP server that connects to this host NE.
primary_pool	TYP_pool	The primary resource pool used by the NEP managing this host NE.
max_connections	TYP_short	The maximum number of concurrent connections allowed to this host NE.
drop_timeout	TYP_short	The maximum inactivity (in minutes) before the NEP drops the primary connection to this host NE.
spawn_threshold	TYP_short	The number of ASDL requests in the SARM ASDL Ready Queue at which point the NEP opens a new auxiliary connection to the destination NE.
kill_threshold	TYP_short	The number of ASDL requests in the SARM ASDL Ready Queue. When the number of requests reaches the kill threshold, the SARM disconnects one or more auxiliary connections.
template_flag	TYP_short	Flag to indicate if this network element entry identifies a static NE (N) or a dynamic network element template (Y).

SSP_list_resource

This function lists NEP resource records from tbl_resource_pool. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-58 SSP_list_resource Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
asap_sys	The ASAP environment (for instance, “TEST", “PROD").	No	I
pool	The pool name.	No	I
device	The physical or logical device name.	No	I

Table 4-59 SSP_list_resource Results

Name	Datatype	Description
asap_sys	TYP_code	The ASAP environment.
pool	TYP_pool	The pool name.
device	TYP_device	The physical or logical device name.
line_type	TYP_dev_type	The device type.
vs_key	TYP_long	Reserved. The shared memory segment identifier for the Virtual Screen buffer.

SSP_list_srp

This function lists SRP definitions from tbl_asap_srp. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-60 SSP_list_srp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srp_id	The logical SRP name.	No	I

Table 4-61 SSP_list_srp Results

Name	Datatype	Description
srp_id	TYP_code	The logical SRP name.
srp_desc	varchar(40)	The SRP description.
srp_conn_type	TYP_srp_conn_type	Connection protocol for the SARM to SRP.
srp_host_name	TYP_host_name	Name of the machine that the SRP resides upon.
srp_host_port	TYP_host_port	The port number that the SRP is listening on for socket connections.
wo_estimate_evt	TYP_code	The work order estimate notification event.
wo_failure_evt	TYP_code	The work order failure notification event.
wo_complete_evt	TYP_code	The work order completion notification event.
wo_start_evt	TYP_code	The work order startup notification event.
wo_soft_err_evt	TYP_code	The work order soft error notification event.
wo_blocked_evt	TYP_code	The work order blocked notification event.
wo_rollback_evt	TYP_code	The work order rollback notification event.
wo_timeout_evt	TYP_code	The work order timeout notification event.
wo_accept_evt	TYP_code	The work order acceptance notification event.
ne_unknown_evt	TYP_code	The unknown NE notification event.

Table 4-61 (Cont.) SSP_list_srp Results

Name	Datatype	Description
ne_avail_evt	TYP_code	The NE available notification event.
ne_unavail_evt	TYP_code	The NE unavailable notification event.
aux_srp_id	TYP_code	The name of the sister SRP.
aux_srp_conn_type	TYP_srp_conn_type	Connection protocol for SARM to auxiliary SRP communications.
aux_srp_host_name	TYP_host_name	Name of the machine that the auxiliary SRP resides upon.
aux_srp_host_port	TYP_host_port	The number of the port that the auxiliary SRP is listening on for socket connections.

SSP_list_stat_text

This function is used to list static text located in tbl_stat_text. This information is retrieved from tbl_stat_text.

For more information about using functions, see "Oracle Execution Examples."

Table 4-62 SSP_list_stat_text Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
stat_id	The logical group of static text messages such as WO_STATE. If this is not specified, all entries in tbl_stat_text are listed.	No	I

Table 4-63 SSP_list_stat_text Results

Name	Datatype	Description
srp_id	TYP_code	The logical SRP name.
status	TYP_stat	Integer key field for grouping.
code	TYP_stat_code	String key field for grouping.
stat_text	TYP_stat_txt	Text describing the label.

SSP_list_user_err_threshold

This function is used to list the user-defined error thresholds for a specific NE, ASDL command, and user error type. This information is retrieved from tbl_user_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-64 SSP_list_user_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
host	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	The ASDL command.	No	I
user_type	The user-defined error type.	No	I

Table 4-65 SSP_list_user_err_threshold Results

Name	Datatype	Description
host_clli	TYP_clli	Host NE.
asdl_cmd	TYP_asdl_cmd	ASDL command.
user_type	TYP_code	User-defined type.
minor_threshold	int	Minor event threshold.
minor_event	TYP_code	System event to be triggered when the minor threshold is reached.
major_threshold	int	Major event threshold.
major_event	TYP_code	System event to be triggered when the major threshold is reached.
critical_threshold	int	Critical event notification threshold.
critical_event	TYP_code	System event to be triggered when the critical threshold is reached.

SSP_list_userid

This function lists user ID definitions from tbl_uid_pwd for the SARM security check logic in the SARM connect handler. You can use this procedure to validate SRP connections to the SARM. Wildcards are allowed.

For more information about using functions, see "Oracle Execution Examples."

Table 4-66 SSP_list_userid Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
uid	The user ID.	No	I

Table 4-67 SSP_list_userid Results

Name	Datatype	Description
uid	TYP_user_id	The user ID.
pwd	TYP_pwd	User password.

Table 4-67 (Cont.) SSP_list_userid Results

Name	Datatype	Description
status	varchar(40)	User's current status.

SSP_ne_monitor

This is an ASAP function.

SSP_new_asdl_defn

This function defines an ASDL configuration record to tbl_asdl_config.

Syntax:

```
exec :retval :=  SSP_new_asdl_defn ('asdl_cmd','reverse_asdl','ignore_rollback',
'route_flag','[description]')
```

Example:

```
exec :retval :=  SSP_new_asdl_defn ('M-CREATE_SINGLE_LINE_ACCESS','M-
DELETE_SINGLE_LINE_ACCESS','N','B','Create a single-line access service.')
```

For more information about using functions, see "Oracle Execution Examples."

This example defines the ASDL command M-CREATE_SINGLE_LINE_ACCESS and routes it to the NEP. If the ASDL command fails, rollback is initiated and the ASDL command **M-DELETE_SINGLE_LINE_ACCESS** is generated using the parameters from the normal ASDL command.

Table 4-68 SSP_new_asdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command.	Yes	I
reverse_asdl	The reverse ASDL to be invoked should this ASDL require rollback. This ASDL command is only issued if the **ignore_rollback** flag is set to 'N'.	Yes	I
ignore_rollback	Specifies whether or not to rollback the ASDL command. If you want rollback to be performed on the ASDL command, set this flag to 'N', as well as specifying the reverse ASDL command.	Yes	I
route_flag	The routing of the ASDL, where: N – Routes the ASDL to the NEP. This is the only valid value.	Yes	I
description	A brief description of the ASDL command.	No	I

SSP_new_asdl_map

This function adds a new ASDL command to a State Table based on the technology and software load in tbl_nep_asdl_prog. The ASDL command must be defined in the configuration records in order for this function to add a new mapping.

Syntax:

```
exec :retval :=  SSP_new_asdl_map ('tech','sftwr_load','asdl_cmd','program',
'interpreter_type')
```

Example:

```
exec :retval :=  SSP_new_asdl_map ('DMS','BCS33','M-CREATE_SINGLE_LINE_ACCESS',
'NEW_LINE_SINGLE','S')
```

This example inserts a mapping for the ASDL command **M-CREATE_SINGLE_LINE_ACCESS** for a DMS switch operating with software load BCS33. For these particular conditions, the Interpreter invokes the State Table program NEW_LINE_SINGLE.

For more information about using functions, see "Oracle Execution Examples."

Table 4-69 SSP_new_asdl_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
tech	The technology type of the NE or SRP with which the Interpreter is to interact.	Yes	I
sftwr_load	The version of the software currently running on the NEP or SRP.	Yes	I
asdl_cmd	The ASDL command.	Yes	I
program	The State Table name.	Yes	I
interpreter_type	A value of 'S' indicates a State Table interpreter, whereas a value of 'J' indicates a JInterpreter. A null value defaults to 'S'.	Yes	I

SSP_new_asdl_parm

This function defines up to nine ASDL parameters for the specified ASDL command starting at base_seq_no in the SARM database parameters.

You cannot add an ASDL command parameter unless the configuration record for the ASDL command has been defined in the SARM database. This ensures data consistency in the static tables.

This function adds new ASDL parameters for a specified ASDL to tbl_asdl_parm.

Syntax:

```
exec :retval :=  SSP_new_asdl_parm ('asdl_cmd','base_seq_no','asdl_parm_lbl',
'csdl_parm_lbl','[def_vlu]','[parm_typ]','asdl_parm_lbl'],'csdl_parm_lbl',
'[def_vlu]','[parm_typ]'...)
```

Example:

In the following example, seven parameters are added to the ASDL command **M-**CREATE_SINGLE_LINE_ACCESS, starting with parameter sequence number 1. All parameters are required, and the **INPA** defaults to 506 if it is not supplied. All other parameters result in a SARM translation error if the parameter is not supplied.

```
exec :retval :=  SSP_new_asdl_parm ('M-CREATE_SINGLE_LINE_ACCESS', 1,
```

```
'NPA', 'INPA', '506', 'R',
'NXX', 'INXX', '', 'R',
'LINE', 'ILINE', '', 'R',
'EXT', 'IEXT', '', 'R',
'LEN', 'ILEN', '', 'R',
'MCLI', 'IMCLI', '', 'R',
'TYPE', 'TYPE', '', 'R'
'SUBSCRIPTION[++]', 'SUBSCRIBER', '', 'C')
```

For more information about using functions, see "Oracle Execution Examples."

Table 4-70 SSP_new_asdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command to add parameters to. You can add up to nine parameters to **asdl_cmd**.	Yes	I
base_seq_no	Starting with a base parameter sequence number (). If you add new parameters to an existing ASDL command, enter the next available sequence number for the **base_seq_no** so that existing parameters are not overwritten.	Yes	I
asdl_parm_lbl	ASDL command parameter is transmitted to the NEP with the parameter value. The type of value for this parameter depends on the parameter format you chose: • **Scalar asdl_parm_lbl** – Specifies the parameter label that is sent on the ASDL command. • **Compound asdl_parm_lbl** – Specifies the base name for the parameter that is sent on the ASDL command. • **Indexed asdl_parm_lbl** – Is the base name for the indexed parameter to be sent on the ASDL command. The current index value is specified using the special literal “++".	Yes	I
csdl_parm_lbl	The parameter label either received from the SRP with the parameter value on the CSDL command or the WO parameter label returned from the NEP State Tables. The label for this parameter depends on the parameter format you choose: • **Scalar csdl_parm_lbl** – Specifies the CSDL command or global WO parameter label. • **Compound csdl_parm_lbl** – Specifies the base name for the compound parameter in the CSDL command or global work order parameter list. • **Indexed csdl_parm_lbl** – Specifies the base name for the indexed parameter in the CSDL command or global work order parameter list. To specify the current index, enter the special literal “++". The initial value is 1 and each ASDL command that is sent causes the value to increment until no more parameters are found at the current index.	Yes	I
def_vlu	The default value for the scalar parameter if it is not specified in the CSDL command or global parameter list. For compound parameters, a default value is not used if a mandatory value can be found.	No	I

Table 4-70 (Cont.) SSP_new_asdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
parm_typ	Indicates the parameter format by using one of the following values: • R – Required scalar parameter • O – Optional scalar parameter • C – Required compound parameter • N – Optional compound parameter • S – Parameter count. This value gives the State Table or Java method the total number of parameters associated with this ASDL command. • I – Optional indexed parameter • M – Mandatory indexed parameter • X – Required XML • Y – Optional XML • P – Required XPATH • Q – Optional XPATH • + – Current index value for the ASDL command parameter. This value is transmitted to the NEP and equals the current index value for the indexed parameter. Note, the previous example makes use of a compound indexed parameter ('SUBSCRIPTION[++]', 'SUBSCRIBER', '', 'C'). For more information on compound indexed parameters, refer to the *ASAP Cartridge Development Guide*.	No	I
al10	Reserved.	No	I

SSP_new_clli_map

This function defines a mapping from a remote CLLI to a host CLLI in tbl_clli_route.

For more information about using functions, see "Oracle Execution Examples."

Table 4-71 SSP_new_clli_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
mach_clli	The remote NE.	Yes	I
host_clli	The host NE identifier of an NE or SRP.	Yes	I
asdl_cmd	The ASDL command.	No	I

SSP_new_comm_param

This function adds a communication parameter for a specified device type, host, and device into tbl_comm_param. The COMMON_HOST_CFG and/or parameter definition can be used for the following combinations:

- common host and common device

- common host and specific device
- specific host and common device
- specific host and specific device

Because the second and third combinations may overlap, the system warns you when communication parameters are updated.

For more information about using functions, see "Oracle Execution Examples."

Table 4-72 SSP_new_comm_param Parameters

Name	Description	Req'd	(I)nput/ (O)utput
dev_type	The device type. Choose from the following connections: • D – Serial Port Dialup • F – TCP/IP FTP Connection • G – Generic Terminal Based Connection • H – Serial Port Hardwired • M – Generic Message Based Connection • P – SNMP Connection • S – TCP/IP Socket Connection • T – TCP/IP Telnet Connection • W – LDAP Connection • C – CORBA	Yes	I
host	The host CLLI. Set to COMMON_HOST_CFG or the host CLLI associated with the command processor. If a common host, the parameter value is the default value, otherwise, it is host-specific.	Yes	I
device	The physical or logical device name. Set to COMMON_DEVICE_CFG, the device associated with the command processor, or the name of a multiplexing device. If a common device, the parameter value is the default value, otherwise, it is device-specific.	Yes	I
param_label	The communication parameter label.	Yes	I
param_value	The communication parameter value.	Yes	I
param_desc	The communication parameter description.	Yes	I

SSP_new_csdl_asdl

This function defines up to nine CSDL-to-ASDL mappings from an CSDL command to an ASDL command with consecutive numbers starting from **base_seq_no** to **tbl_csdl_asdl**.

CSDL and ASDL commands must be defined before adding a CSDL-to-ASDL mapping relationship, otherwise this function rejects the insertion attempt. This is an enforced integrity check that ensures configuration consistency. You can add up to nine mapping relationships with one procedure call.

Syntax:

```
exec :retval :=  SSP_new_csdl_asdl ('csdl_cmd','base_seq_no',
'asdl_cmd','cond_flag','parm_lbl','parm_vlu','eval_exp','prn',
'[asdl_cmd]','[cond_flag]','[parm_lbl]','[parm_vlu]','[eval_exp]',

'[prn]', ...)
```

Example:

```
exec :retval :=  SSP_new_csdl_asdl ('M-CREATE_BUS_LINE', 1,

'M-CLEAR_INTERCEPT', 'A', '', '', '', '0',
'M-CREATE_SINGLE_LINE_ACCESS', 'A', '', '', '', '0',
'ADD_ALWAYS_ON_3WC', 'D', 'ALWAYS_ON_AREA', '', '', '0',
'ADD_ALWAYS_ON_CRT', 'D', 'ALWAYS_ON_AREA', '', '', '0',
'ADD_ALWAYS_ON_CTR', 'A', '', '','' '0')
```

In this example, the CSDL command **M-CREATE_BUS_LINE** is mapped to four ASDL commands that start with the ASDL sequence number 1.

ASDL commands **M-CLEAR_INTERCEPT** and **M-CREATE_SINGLE_LINE_ACCESS** are always generated.

ADD_ALWAYS_ON_3WC and **ADD_ALWAYS_ON_CRT** are generated only if the CSDL command parameter **ALWAYS_ON_AREA** is defined.

The fifth ASDL command, **ADD_ALWAYS_ON_CTR**, is generated only if the CSDL command parameter **TRACE_OPT** is defined and has an ON value.

For more information about using functions, see "Oracle Execution Examples."

Table 4-73 SSP_new_csdl_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_cmd	The CSDL command identifier.	Yes	I
base_seq_no	The number of the first ASDL mapping to insert. Because each CSDL command may map to several ASDL commands, an index is kept in the static table and used to determine the order of the ASDL commands being sent to the NEP. Up to nine ASDL commands can be mapped with a single call of this procedure. Each call is assigned a sequence number based on the **base_seq_no**. For example, if the CSDL command maps to twelve ASDL commands, the base sequence number should be 1 for the first procedure call (which will create nine mapping relationships, ASDL command sequence numbers 1-9) and 10 for the second procedure call (which will create the remaining three mapping relationships, ASDL command sequence numbers 10-12).	Yes	I
asdl_cmd	The name of the ASDL command that the CSDL command maps to. The ASDL command is generated by the CSDL command based on the value of the condition flag (**cond_flag**).	Yes	I

Table 4-73 (Cont.) SSP_new_csdl_asdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
cond_flag	Used to specify conditions that need to be met in order for the SARM to generate the ASDL command for the CSDL command. Type one of the following values: • **A** – Always generates the ASDL command for the CSDL command • **D** – Generates the ASDL command if the CSDL parameter is defined (present) • **N** – Generates the ASDL command if the CSDL parameter is not defined (present) • **E** – Generates the ASDL command if the CSDL parameter is defined and equal to a value. The generation of each ASDL command depends upon the results of the previous ASDL. When the previous command completes successfully, it returns parameters to the SARM. When using 'cond_flag'='E', the following values are required: • 'lbl1' • 'lbl2' • 'val1' • 'val2' When using 'cond_flag'='D' or 'N', the following values are required: • 'lbl1' • 'lbl2'	Yes	I
parm_lbl and parm_vlu parameters	Required when you use CSDL parameter-dependent conditions. Set the CSDL command parameter name for 'D', 'N', and 'E' condition flags using **parm_lbl**. The 'E' condition flag checks that the CSDL command parameter is equal to the value specified by **parm_vlu**. For more information about these condition flags, refer to the previous parameter, **cond_flag**.	Yes	I
eval_exp	Contains combination of parameter names, operators, and values to which the parameters are compared.	No	I
pnr	Value of 'point of no return' for rollbacks. Values are: • 0 (default) – This ASDL is not the 'point of no return' for rollback purposes • 1 – This ASDL is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this ASDL but no further. • 2 – 'point of no return' for no rollback. Once past this ASDL, no rollback can occur.	Yes	I

SSP_new_csdl_asdl_idx

This function allows multiple conditions to be inserted into **tbl_csdl_asdl_eval**. Up to nine rules can be inserted with each call. If adding up to nine rules, leave append_rule set to 0. To add more than nine rules to one mapping, call the function again with append_rule set to 1.

Syntax:

```
exec :retval :=  SSP_new_csdl_asdl_idx (‘append_rule’,‘csdl_cmd’, ‘base_seq_no’,
‘asdl_cmd’, c‘cond_flag’, ‘parm_lbl’, ‘parm_vlu’,
‘eval_exp’, ‘apply_from’, ‘apply_to’,
‘[cond_flag]’, ‘[parm_lbl]’, ‘[parm_vlu]’, ‘[eval_exp]’,‘[apply_from]’,
‘[apply_to])
...
```

Examples:

```
exec :retval :=  SSP_new_csdl_asdl_idx (0,'M-CREATE_BUS_LINE',1,'M-
CLEAR_INTERCEPT','0',
'A', '', '', '', 1, 1)

exec :retval :=  SSP_new_csdl_asdl_idx (0,'M-CREATE_BUS_LINE',2,
'M-CREATE_SINGLE_LINE_ACCESS', '0',

'A', '', '', '', 1, 1)


exec :retval :=  SSP_new_csdl_asdl_idx (0,'M-
CREATE_BUS_LINE',3,'ADD_ALWAYS_ON_3WC','0',

'D', 'ALWAYS_ON_AREA', '', '', 1, 1)


exec :retval :=  SSP_new_csdl_asdl_idx (0,'M-
CREATE_BUS_LINE',4,'ADD_ALWAYS_ON_CRT','0',
'D', 'ALWAYS_ON_AREA', '', '', 1, 1)

exec :retval :=  SSP_new_csdl_asdl_idx (0,'M-
CREATE_BUS_LINE',5,'ADD_EMAIL_ACCOUT','0',

'E', 'MAX_ADDRESS_EXCEED', 'FALSE', '', 1, 9999,
'E', 'CREATE_ADDRESS_1', 'TRUE', '', 1, 1,
'E', 'CREATE_ADDRESS_2', 'TRUE', '', 2, 2,
'E', 'CREATE_ADDRESS_3', 'TRUE', '', 3, 3,
'E', 'CREATE_ADDRESS_4', 'TRUE', '', 4, 4,
'E', 'CREATE_ADDRESS_5', 'TRUE', '', 5, 5,
'E', 'CREATE_ADDRESS_6', 'TRUE', '', 6, 6,
'E', 'CREATE_ADDRESS_7', 'TRUE', '', 7, 7,
'E', 'CREATE_ADDRESS_8', 'TRUE', '', 8, 8)


exec :retval :=  SSP_new_csdl_asdl_idx (1,'M-
CREATE_BUS_LINE',5,'ADD_EMAIL_ACCOUT','0',

'E', 'CREATE_ADDRESS_9', 'TRUE', '', 9, 9,
'E', 'CREATE_ADDRESS_10', 'TRUE', '', 10, 10,
'E', 'CREATE_ADDRESS_11', 'TRUE', '', 11, 11)
```

In this example, the CSDL command **M-CREATE_BUS_LINE** is mapped to five ASDL commands that start with the ASDL sequence number 1.

ASDL commands **M-CLEAR_INTERCEPT** and **M-CREATE_SINGLE_LINE_ACCESS** are always generated.

ADD_ALWAYS_ON_3WC and **ADD_ALWAYS_ON_CRT** are generated only if the CSDL command parameter **ALWAYS_ON_AREA** is defined.

The fifth ASDL command, **ADD_EMAIL_ACCOUNT**, is an indexed ASDL command that contains 12 rules. Each instance of this ASDL command is generated only if the CSDL command parameter **MAX_ADDRESS_EXCEED** is defined and the current index rule has a **FALSE** value and for each index rule is evaluated to true.

For example, if the current index is 5 then this rule ('E', 'CREATE_ADDRESS_5', 'TRUE', '', 5, 5) says the CSDL command parameter CREATE_ADDRESS_5 is defined and the current index rule has a TRUE value.

Note that the last call of the fifth example has **append_rule** set to 1 to indicate that the rules are to be appended to those added by the previous call:

```
exec :retval :=  SSP_new_csdl_asdl_idx (1,'M-CREATE_BUS_LINE',5...)
```

For more information about using functions, see "Oracle Execution Examples."

Table 4-74 SSP_new_csdl_asdl_idx Parameters

Name	Description	Req'd	(I)nput/ (O)utput
append_rule	Indicates whether rules are inserted to a new or existing mapping. The initial call, with this parameter set to 0, can insert up to 9 rules. To add additional rules, call SSP_new_csdl_asdl_idx again, with this parameter set to 1.	Yes	I
csdl_cmd	The CSDL command identifier.	Yes	I
base_seq_no	The number of the first ASDL mapping to insert. Because each CSDL command may map to several ASDL commands, an index is kept in the static table and used to determine the order of the ASDL commands being sent to the NEP. Up to nine ASDL commands can be mapped with a single call of this procedure. Each call is assigned a sequence number based on the **base_seq_no**. For example, if the CSDL command maps to twelve ASDL commands, the base sequence number should be 1 for the first procedure call (which will create nine mapping relationships, ASDL command sequence numbers 1-9) and 10 for the second procedure call (which will create the remaining three mapping relationships, ASDL command sequence numbers 10-12).	Yes	I
asdl_cmd	The name of the ASDL command that the CSDL command maps to. The ASDL command is generated by the CSDL command based on the value of the condition flag (**cond_flag**).	Yes	I

Table 4-74 (Cont.) SSP_new_csdl_asdl_idx Parameters

Name	Description	Req'd	(I)nput/ (O)utput
pnr	Value of 'point of no return' for rollbacks. Values are: • 0 (default) – This ASDL is not the 'point of no return' for rollback purposes • 1 – This ASDL is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this ASDL but no further. • 2 – 'point of no return' for no rollback. Once past this ASDL, no rollback can occur.	Yes	I
cond_flag	Used to specify conditions that need to be met in order for the SARM to generate the ASDL command for the CSDL command. Type one of the following values: • **A** – Always generates the ASDL command for the CSDL command • **D** – Generates the ASDL command if the CSDL parameter is defined (present) • **N** – Generates the ASDL command if the CSDL parameter is not defined (present) • **E** – Generates the ASDL command if the CSDL parameter is defined and equal to a value. The generation of each ASDL command depends upon the results of the previous ASDL. When the previous command completes successfully, it returns parameters to the SARM. When using 'cond_flag'='E', the following values are required: • 'lbl1' • 'lbl2' • 'val1' • 'val2' When using 'cond_flag'='D' or 'N', the following values are required: • 'lbl1' • 'lbl2'	Yes	I
parm_lbl parm_vlu	Required when you use CSDL parameter-dependent conditions. Set the CSDL command parameter name for 'D', 'N', and 'E' condition flags using **parm_lbl**. The 'E' condition flag checks that the CSDL command parameter is equal to the value specified by **parm_vlu**. For more information about these condition flags, refer to the previous parameter, **cond_flag**.	Yes	I
eval_exp	Contains combination of parameter names, operators, and values to which the parameters are compared.	No	I

Table 4-74 (Cont.) SSP_new_csdl_asdl_idx Parameters

Name	Description	Req'd	(I)nput/ (O)utput
apply_from	The first indexed ASDL that this rule should apply to: Valid range is from 1 to 9999. Must be less than or equal to the value specified in column apply_to. If is not specified, then this rule will be applied to any indexed ASDL up to and including the one specified in column apply_to.	No	I
apply_to	The last indexed ASDL that this rule should apply to: Valid range is from 1 to 9999. Must be greater than or equal to the value specified in column apply_from. If is not specified, then this rule will be applied to any indexed ASDL starting from the one specified in column apply_from.	No	I

SSP_new_csdl_defn

This function adds a new CSDL command into tbl_csdl_config.

Syntax:

```
exec :retval :=  SSP_new_csdl_defn ('csdl_cmd', 'rollback_req', csdl_level,
'[fail_event]', '[complete_event]', '[description]')
```

Example:

```
exec :retval :=  SSP_new_csdl_defn ('M-CREATE_BUS_LINE', 'Y', 82, 'SYS_ERR',
'SYS_INFO', 'Add Business Access Line')
```

In this example, the **M-CREATE_BUS_LINE** CSDL command adds a business access line with a CSDL command level of 82. If the CSDL command fails, the **SYS_ERR** system event is triggered and rollback is performed on the entire order. Upon successful completion, the **SYS_INFO** system event is issued.

For more information about using functions, see "Oracle Execution Examples."

Table 4-75 SSP_new_csdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_cmd	The name of the CSDL command to add. It should be a unique CSDL command label in ASAP.	Yes	I
rollback_req	A Yes/No (Y or N) flag that indicates whether rollback is required for this CSDL command. If you set this flag to Y, ASAP automatically rolls back any actions performed by the work order if the work order fails.	Yes	I

Table 4-75 (Cont.) SSP_new_csdl_defn Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_level	An integer between 0 and 255 that indicates the sequence level for the CSDL command within the work order. The SARM uses this integer to determine the order in which to provision CSDL commands from an SRP and then provisions CSDL commands that have lower level numbers first. Sequence levels are only relevant for inter-dependent CSDL commands.	Yes	I
fail_event	ASAP system events that are triggered upon completion or failure of the CSDL command. The events must be first defined in the control database if alarms are to be generated from such events. These are optional parameters.	No	I
complete_event		No	I
description	A CSDL command description. ASAP front-end tools that are monitoring the progress of the work order can use this description. This is an optional parameter.	No	I

SSP_new_dn_map

This function adds new ASDL command routings by directory number to tbl_nep_rte_asdl_nxx.

For more information about using functions, see "Oracle Execution Examples."

Table 4-76 SSP_new_dn_map Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asdl_cmd	The ASDL command.	No	I
npa	The Numbering Plan Area code.	No	I
nxx	The Central Office code.	No	I
from_line	The lowest line number in the range of telephone numbers to provide routing for.	No	I
to_line	The highest line number in the range.	No	I
queue_nm	The host NE to which this ASDL should be routed.	Yes	I

SSP_new_id_routing

This function adds a new host NE and the ID_ROUTING mapping record to tbl_id_routing. You can use this function when routing by ID_ROUTING is used.

For more information about using functions, see "Oracle Execution Examples."

Table 4-77 SSP_new_id_routing Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier.	No	I
asdl_cmd	The ASDL command.	Yes	I
id_routing_from	The starting point of a range of ID_ROUTING.	No	I
id_routing_to	The end point of a range of ID_ROUTING.	No	I

SSP_new_intl_msg

This function defines an international message for a particular language in tbl_msg_convert.

For more information about using functions, see "Oracle Execution Examples."

Table 4-78 SSP_new_intl_msg Parameters

Name	Description	Req'd	(I)nput/ (O)utput
lang_cd	The language code.	Yes	I
msg_id	The unique message identifier.	Yes	I
type	The type of message formatting.	Yes	I
message	The message text.	Yes	I
var_description	The description of the substitutable fields, if any, within the message.	No	I
wo_audit	Destination for the log message.	No	I

The following example shows how to add an international message:

```
exec :retval :=  SSP_new_intl_msg ('USA', 1, 'D', 'Work Order %s Timed Out',  'WO Id: %s')
```

This example adds an international message to the SARM database for American English (USA). International messages use parameters to identify the entity they are associated with. The **var_description** parameter ('WO id: %s') specifies the format and the arguments that are used to generate the actual message.

SSP_new_ne_host

This function defines a host NE with its technology type, software version, and inventory manager in the SARM database table, tbl_host_clli.

For more information about using functions, see "Oracle Execution Examples."

Table 4-79 SSP_new_ne_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE to which the remote NE is connected.	Yes	I

Table 4-79 (Cont.) SSP_new_ne_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
tech_type	The technology type of the host NE or SRP.	Yes	I
sftwr_load	The version of the software currently running on the NEP or SRP.	Yes	I

SSP_new_nep

This function defines a secondary (dialup) pool of devices or connections for a specified NEP in the SARM database. This function adds a pool of devices or connections to tbl_nep.

For more information about using functions, see "Oracle Execution Examples."

Table 4-80 SSP_new_nep Parameters

Name	Description	Req'd	(I)nput/ (O)utput
nep_svr_cd	The NEP managing the secondary pool of devices.	Yes	I
dialup_pool	The secondary pool of devices.	No	I

SSP_new_nep_program

This function inserts or updates a line of State Table code into tbl_nep_program. If the line exists, it will be updated.

If there is no ASDL-to-State Table mapping relationship, the user is warned that the mapping relationship does not exist. The insertion of the State Table, however, is not affected.

For more information about using functions, see "Oracle Execution Examples."

Table 4-81 SSP_new_nep_program Parameters

Name	Description	Req'd	(I)nput/ (O)utput
program	The State Table program identifier.	Yes	I
line_no	The State Table line number to delete. If set to NULL, all lines of the State Table are deleted.	Yes	I
action	The action string identifying a particular action performed by the Interpreter.	Yes	I
act_string	The action string associated with the State Table.	Yes	I
act_int	The action integer.	Yes	I

SSP_new_net_elem

This function defines a host NE in the SARM database (tbl_ne_config).

For more information about using functions, see "Oracle Execution Examples."

Table 4-82 SSP_new_net_elem Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	Yes	I
nep_svr_cd	The logical name of the NEP that connects to this host NE.	Yes	I
primary_pool	The primary resource pool used by the NEP managing this host NE to determine the devices to use to connect to it.	Yes	I
max_connections	The maximum number of concurrent connections allowed to this host NE.	Yes	I
drop_timeout	The maximum inactivity (in minutes) before NEP drops the primary connection to this host NE.	Yes	I
spawn_threshold	Number of ASDL requests in the SARM ASDL Ready Queue to be exceeded before the NEP opens a new auxiliary connection to that NE.	Yes	I
kill_threshold	Once the SARM has fewer ASDL requests in its ASDL Ready Queue than this number, it disconnects one or more auxiliary connections.	Yes	I
template_flag	Flag to indicate whether this network element entry identifies a static NE (N) or a dynamic network element template (Y).	Y	I

SSP_new_resource

This function defines an NEP resource (“device") to be used for NE access in the SARM database (tbl_resource_pool).

For more information about using functions, see "Oracle Execution Examples."

Table 4-83 SSP_new_resource Parameters

Name	Description	Req'd	(I)nput/ (O)utput
asap_sys	The ASAP environment (TEST, PROD, etc.)	Yes	I
pool	The pool name.	Yes	I
device	The physical or logical device name.	Yes	I
line_type	The type of line for the serial communication.	Yes	I
vs_key	Reserved. The shared memory segment identifier for the Virtual Screen buffer.	No	I

SSP_new_srp

This function adds an SRP to tbl_asap_srp.

For more information about using functions, see "Oracle Execution Examples."

Table 4-84 SSP_new_srp Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srp_id	The logical SRP name.	Yes	I
srp_desc	The SRP description.	Yes	I
aux_srp_id	The name of the sister SRP.	No	I
wo_estimate_evt	The work order estimate notification event.	No	I
wo_failure_evt	The work order failure notification event.	No	I
wo_complete_evt	The work order completion notification event.	No	I
wo_start_evt	The work order startup notification event.	No	I
wo_soft-err_evt	The work order soft error notification event.	No	I
wo_blocked_evt	The work order blocked notification event.	No	I
wo_rollback_evt	The work order rollback notification event.	No	I
wo_timeout_evt	The work order timeout notification event.	No	I
ne_unknown_evt	The unknown NE notification event.	No	I
ne_avail_evt	The NE available notification event.	No	I
ne_unavail_evt	The NE available notification event.	No	I
wo_accept_evt	The system event to be issued.	No	I
srp_conn_type	Connection protocol for SARM to SRP.	No	I
srp_host_name	SRP host machine name.	No	I
srp_host_port	Port number for socket connections.	No	I
aux_srp_conn_type	Connection protocol for SARM communication to the auxiliary SRP.	No	I
aux_srp_host_name	Host machine name of the auxiliary SRP.	No	I
aux_srp_host_port	Port number for socket connections on an auxiliary SRP.	No	I

SSP_new_stat_text

This function adds new static text into tbl_stat_text. If an entry already exists for the static text identifier, the static text is updated with the new information.

For more information about using functions, see "Oracle Execution Examples."

Table 4-85 SSP_new_stat_text Parameters

Name	Description	Req'd	(I)nput/ (O)utput
stat_id	The logical group of static text messages.	Yes	I
status	The integer identifier for member of a logical grouping.	No	I

Table 4-85 (Cont.) SSP_new_stat_text Parameters

Name	Description	Req'd	(I)nput/ (O)utput
code	The string identifier for a member of a logical grouping.	No	I
stat_text	The actual text message to use in place of a string/ integer identifier.	Yes	I

SSP_new_user_err_threshold

This function creates a new user-defined error threshold in the system for the specified NE, ASDL command, and the user-defined error type in tbl_user_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-86 SSP_new_user_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	Yes	I
asdl_cmd	The ASDL command.	Yes	I
user_type	The user-defined error type.	Yes	I
minor_threshold	The threshold for minor system events. This is the number of times the user_type can be returned before the corresponding minor event is generated.	Yes	I
minor_event	The minor system event to be generated when the threshold is exceeded.	Yes	I
major_threshold	The threshold for major system events. This is the number of times the user_type can be returned before the corresponding major event is generated.	Yes	I
major_event	The major system event to be generated when the major threshold is reached.	Yes	I
critical_threshold	The threshold for critical event notifications. This is the number of times the user_type can be returned before the corresponding critical event is generated.	Yes	I
critical_event	The critical system event to be generated when the critical threshold is reached.	Yes	I

SSP_new_userid

This function adds a new user account for the SARM to control access from the SRP in tbl_uid_pwd.

For more information about using functions, see "Oracle Execution Examples."

Table 4-87 SSP_new_userid Parameters

Name	Description	Req'd	(I)nput/ (O)utput
uid	The user ID.	Yes	I
pwd	The password.	Yes	I
status	The user's current status.	No	I

SSP_orphan_purge

This stored procedure scans SARM database tables and deletes fragments of old work orders that do not have an entry in tbl_wrk_ord. Occasionally, the database becomes fragmented and records are left behind in various tables, including:

- tbl_asap_stats
- tbl_info_parm
- tbl_srq
- tbl_srq_csdl
- tbl_srq_log
- tbl_asdl_log
- tbl_srq_parm
- tbl_srq_asdl_parm
- tbl_wo_event_queue

This stored procedure is time-consuming and requires considerable system resources. Therefore, it should not run during peak hours.

For more information, see Database Purging in the *ASAP System Administrator's Guide*.

For more information about using functions, see "Oracle Execution Examples."

Error management

The management of errors related to provisioning by an NEP provides a detailed error tracking scheme and lets the administrator configure error-processing thresholds using NE and ASDL commands. The thresholds control the release of specific ASDL commands to the NE to prevent an excessive number of errors from occurring.

The following table lists the types or errors that can occur:

- **SUCCEED:** The ASDL provisioning was successful.
- **FAIL:** Fails the current order and stops any subsequent processing.
- **RETRY**: Retries the current ASDL command after a user-configured interval and up to a user-configured number of times before failing the order.
- **MAINTENANCE:** Causes the current ASDL command to wait for the NE to come out of maintenance before processing continues.
- **SOFT_FAIL:** An error has occurred at the NE but order processing can continue.

- **DELAYED_FAIL**: An ASDL had failed during provisioning. The SARM skips any subsequent ASDL in the CSDL, continues provisioning at the next CSDL, and then fails the order.

Refer to the *ASAP Cartridge Development Guide* for more detailed descriptions of these base_types.

User-configured history windows and polling intervals that update the ASAP database are also supported. Information is available in real-time from the SARM server or in a batch from the ASAP database. This batch information can then be used by administrative tools to perform root cause analysis.

SSP_del_err_threshold

This function deletes error thresholds for a specific NE and ASDL command from tbl_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-88 SSP_del_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	The ASDL command associated with the threshold. This can be NULL to indicate an NE threshold.	No	I

SSP_del_err_type

This function deletes mappings between base and user exit types. These mappings are defined in tbl_user_err.

For more information about using functions, see "Oracle Execution Examples."

Table 4-89 SSP_del_err_type parameters

Name	Description	Req'd	(I)nput/ (O)utput
user_type	The user-defined error type.	Yes	I
asdl	The ASDL that is executing. Error types can be defined for user_type and ASDL combinations.	No	I
csdl	The CSDL that is executing. Error types can be defined for user_type and CSDL combinations.	No	I
ne_vendor	The vendor of the network element.	No	I
tech_type	The technology of the network element.	No	I
sftwr_load	Software version of the host network element.	No	I

SSP_err_enable

This function enables the provisioning of an ASDL command that has been disabled because it exceeded an error threshold. If the NE is down, the NE will be enabled by this function. A

particular ASDL may also be marked as disabled to an NE, therefore it may be re-enabled to that NE by specifying it along with the NE in the call to SSP_err_enable.

> **Note:**
>
> The action performed by this function is not persistent. That is, if the SARM is taken down after this function has been executed, the changes made to the state of the NE and ASDL are lost.

For more information about using functions, see "Oracle Execution Examples."

Table 4-90 SSP_err_enable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	Yes	I
asdl_cmd	The optional ASDL command to enable a specific type of provisioning.	No	I

SSP_list_err_host

This function lists the NEs and the ASDL commands that have been disabled for provisioning.

For more information about using functions, see "Oracle Execution Examples."

Table 4-91 SSP_list_err_host Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
host_clli	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	The ASDL command.	No	I

Table 4-92 SSP_list_err_host Results

Name	DataType	Description
host_clli	TYP_clli	Host NE.
disable_dts	datetime	Timestamp when the NE was disabled.
asdl_cmd	TYP_asdl_cmd	ASDL command that has been disabled.
order_count	TYP_long	Number of orders waiting for the NE.

SSP_list_err_threshold

This function lists the error thresholds for a specific NE and ASDL command. Error thresholds are stored in tbl_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-93 SSP_list_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
host_clli	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	The ASDL command associated with the threshold. This can be NULL to indicate an NE threshold.	No	I

Table 4-94 SSP_list_err_threshold Results

Name	DataType	Description
host_clli	varchar(64)	The host NE identifier.
asdl_cmd	varchar(30)	The ASDL command associated with the threshold. This can be NULL to indicate an NE threshold.
threshold	TYP_long	Error threshold.

SSP_list_err_type

This function lists the mapping between user exit types and base exit types. This mapping is stored in tbl_user_err.

You cannot define both user_type and base_type at the same time.

For more information about using functions, see "Oracle Execution Examples."

Table 4-95 SSP_list_err_type Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
user_type	The user-defined error type.	No	I
base_type	The base error type.	No	I

Table 4-96 SSP_list_err_type Results

Name	DataType	Description
user_type	TYP_code	User-defined type.
base_type	TYP_code	Base error type.
description	varchar(50)	A brief description of the user exit type.

SSP_new_err_threshold

This function adds a new threshold for a specific NE and ASDL command in tbl_err_threshold.

For more information about using functions, see "Oracle Execution Examples."

Table 4-97 SSP_new_err_threshold Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier.	Yes	I
asdl_cmd	The ASDL command associated with the threshold. This can be NULL to indicate an NE threshold.	Yes	I
threshold	The error threshold for the time period (consecutive number of errors before connection to an NE should be disabled).	Yes	I

SSP_new_err_type

This function adds a new mapping between user exit types and the base exit types in tbl_user_err.

For more information about using functions, see "Oracle Execution Examples."

Table 4-98 SSP_new_err_type Parameters

Name	Description	Req'd	(I)nput/ (O)utput
user_type	The user-defined error type.	Yes	I
base_type	The base error type.	Yes	I
description	A brief description of the ASDL command.	No	I
asdl	The ASDL that is executing. Error types can be defined for user_type and ASDL combinations.	No	I
csdl	The CSDL that is executing. Error types can be defined for user_type and CSDL combinations.	No	I
ne_vendor	The vendor of the network element.	No	I
tech_type	The technology of the network element.	No	I
sftwr_load	Software version of the host network element.	No	I
search_pattern	Regular expression pattern that is used to match on network element responses.	No	I

Switch blackout processing

If ASAP shared a port to an NE with another system or if regular NE maintenance must be performed, you can define the NE blackout period during which time the NEP will not connect to that NE.

To identify switch blackout periods, ASAP checks a database table to see if the current time is within the user-defined blackout period. You can configure both static (keyed by date and time) and dynamic (keyed by specific day and time) blackout periods. If a blackout period is detected, the switch is placed into maintenance mode automatically.

SSP_add_blackout

This function configures the static and dynamic blackout periods for a specific NE host. Blackout information is stored in tbl_blackout.

For more information about using functions, see "Oracle Execution Examples."

Table 4-99 SSP_add_blackout Parameters

Name	Description	Req'd	(I)nput/ (O)utput
dayname	The name of the day of the week for a weekly blackout (such as Mondays). Set to NULL to use specific date and time intervals for blackout.	No	I
host_clli	The Host NE identifier of an NE or SRP.	Yes	I
start_tm, end_tm	The start time and end time for the blackout interval. If you have specified a dayname blackout, the blackout is based on the day and the start time and end time. If the dayname parameter is set to NULL, the blackout is based on a specified date and time.	Yes	I
descr	A brief description of the blackout.	No	I

SSP_check_blackout

This function determines whether or not the specified NE is currently blacked out. An NE is blacked out if an entry exists in **tbl_blackout** for the specified NE, where the current time is between the entry's start time and end time.

For more information about using functions, see "Oracle Execution Examples."

Table 4-100 SSP_check_blackout Parameters

Name	Description	Req'd	(I)nput/ (O)utput
curday	The current day.	Yes	I
host_clli	The Host NE identifier.	Yes	I
curr_dt_tm	The current time.	Yes	I
ret	The return status.	Yes	O

SSP_del_blackout

This procedure removes blackout periods for a specific NE host from tbl_blackout.

For more information about using functions, see "Oracle Execution Examples."

Table 4-101 SSP_del_blackout Parameters

Name	Description	Req'd	(I)nput/ (O)utput
dayname	The name of the day of the week for default setup (e.g., Monday). Set to NULL to use specific day and time intervals for the blackout.	No	I
host_clli	The host NE identifier of an NE or SRP.	No	I
start_tm	The start time for the blackout interval. If the dayname parameter is not NULL, then these fields are used as time intervals for the day. If the dayname parameter is NULL, this field must include both date and time.	No	I

SSP_list_blackout

This procedure lists blackout periods for a specific NE host. This information is stored in tbl_blackout.

For more information about using functions, see "Oracle Execution Examples."

Table 4-102 SSP_list_blackout Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
RC2	Oracle Database Ref Cursor.	Yes	I/O
RC3	Oracle Database Ref Cursor.	Yes	I/O
dayname	The name of the day of the week for default setup (e.g., Monday). Set to NULL to use specific date and time intervals for the blackout.	No	I
host_clli	The host NE identifier of an NE or SRP.	No	I
start_tm	The start time for the blackout interval. If the dayname parameter is not NULL, then these fields are used as time intervals for the day. If the dayname parameter is NULL, this field must include both date and time.	No	I

Table 4-103 SSP_list_blackout Results

Name	Datatype	Description
dayname	varchar(10)	The name of the day.
host_clli	TYP_clli	The host CLLI.
start_tm	datetime	The start time for the blackout interval.
end_tm	datetime	The end time for the blackout interval.
description	varchar(40)	The description of the blackout period.
dayname	varchar(10)	The name of the day.
host_clli	TYP_clli	The host CLLI.

Table 4-103 (Cont.) SSP_list_blackout Results

Name	Datatype	Description
start_tm	datetime	The start time for the blackout interval.
end_tm	datetime	The end time for the blackout interval.
description	varchar(40)	The description of the blackout period.
dayname	varchar(10)	The name of the day.
host_clli	TYP_clli	The host CLLI.
start_tm	datetime	The start time for the blackout interval.
end_tm	datetime	The end time for the blackout interval.
description	varchar(40)	The description of the blackout period.

Switch direct interface (SWD)

This section describes ASAP's support of direct terminal session access to NEs (SWDs).

The following assumptions apply:

- Only one SWD session is allowed per NE.
- ASAP automatically selects the port that is used in the session.
- An SWD Client does not have access to the NE if the NE is in maintenance mode or provisioning black-out mode.
- An SWD request has the highest priority in ASAP, but an active ASDL command is not pre-empted.
- The activity that occurs during the session is not logged by ASAP.
- BSD style sockets are used on the server side.
- The SWD Client Application sends/receives only raw data to/from ASAP because ASAP just acts as a gateway for passing data to and from the NE. The SWD Client Application is responsible for mapping raw data received from ASAP to terminal emulation: specific data and vice versa. The SWD client must support the terminal emulation interface used by the NE (for example, VT220).
- Security validation checks for SWD access are controlled by the SWD client.
- When an SWD session is completed, ASAP disconnects from the NE before resuming normal provisioning activities.
- No message is sent to the SWD client if the connection cannot be granted immediately.
- An NEP supports SWD sessions only if it supports Multi-Protocol Manager functionality (that is, it links in the ASAP Communication API **libasccomm**).
- SWD sessions to an NE are possible only if the communication to the NE is terminal-based, not message-based.
- As the number of SWD sessions increases, performance may degrade because all the SWD sessions are managed by a single thread. This arrangement ensures that SARM's primary function of provisioning is not affected by SWD access.

Configuration parameters

Table 4-104 SWD Configuration parameters

Name	Default Value	Cfg File	Description
SWD_LISTEN_PORT	N/A	Global	TCP/IP listen port for the SARM server.
SWD_IDLE_TIMEOUT	120	Global	Idle period in seconds for automatically disconnecting the SWD session if no activity is detected.

General message format

Table 4-105 SWD General message format

Field Name	Offset	Length	Description
Message Type	0	4	Integer message type to be processed.
Data Length	4	4	Integer message data length.
Message Data	8	variable	Optional message data. The format is dependent on message type and is specified below for each type.

All fields are passed using the standard network independent format (i.e., network byte ordering).

SWD Client-to-SARM messages

Table 4-106 SWD Client-to-SARM messages

Msg Type	Msg Format	Description
1	HOST=<hostname> ; USERID=<userid>	Connect to the specified NE for an SWD session (SWD_CLT_\CONNECT).
2	keystroke data	User entered keystroke data. This is binary data (SWD_CLT_DATA).
3		User entered break key (SWD_CLT_BREAK_KEY).
4		SWD session complete (SWD_CLT_NE_DISCONNECT).

SARM-to-SWD client messages

Table 4-107 SARM-to-SWD client messages

Msg Type	Msg Format	Description
10	STATUS=<2-digit status value>; ERR_MSG=<error message>	Connection acknowledgment message (SWD_SESS_CONNECT_ACK).

Table 4-107 (Cont.) SARM-to-SWD client messages

Msg Type	Msg Format	Description
11	session data	Session data returned by the NE (SWD_SESS_DATA).
12		Unexpected port disconnection (SWD_SESS_NE_PORT_FAILURE).
13		SWD session timeout (SWD_SESS_TIMEOUT).

The following table lists all of the valid availability status values and the associated error message formats for message type 10.

Table 4-108 Message Type 10 Information

Availability Status	Error Msg Description	Error Msg Format
1	SWD Session to Host established successfully.	Connected successfully to Host.
2	Time out waiting for NE Connection Request from SWD Client.	Timed out waiting for SWD Connect Request.
3	ASAP not configured to connect to NE.	Invalid Host CLLI.
4	NE does not support SWD Session.	NEP does not support SWD Sessions
5	NEP in loopback mode.	NEP in loopback mode.
6	SWD Session to NE already attempted or in use.	SWD Session to NE already exists or attempted.
7	NEP connecting to NE is down.	NEP down.
8	NE is down.	NE down.
9	NEP connecting to NE is in maintenance mode.	NE in maintenance or blackout mode.
10	NE is disabled.	NE is disabled.
11	NE is busy provisioning for more than the “SWD_CONNECT_WAIT_TIMEOUT " seconds.	Timed out waiting for connection. NE is busy provisioning.
12	Resource Allocation Error; Cannot connect to NEP.	System Resource Error.
13	Not connected yet.	Data/Break/Disconnect request from SWD Client rejected because NE Connect Request not received yet.
14	Connected already.	NE Connect Request rejected since the SWD Client is already connected to an NE.

Stop work order interface

The Stop Work Order feature is a user-generated event from the OCA client or the JSRP that is received directly by the SARM and applied to a specific work order. The event is received as a function **aims_stop_wo**.

Syntax:

```
CS_RETCODE aims_stop_wo_rpc(SRV_PROC *srvproc)
{
    ASAP_WO_ID wo_id;
    CS_INT rollback;
    CS_RETCODE ret_status;
    CS_CHAR tmp[ASAP_SRQ_EVENT_TEXT_L],evt_text[ASAP_SRQ_EVENT_TEXT_L];
    CS_CHAR audit_flag[10];
    CS_CHAR user_id[32];
```

The **aims_stop_wo** function stops a work order that is in progress. This allows the user to correct any problems that may be occurring before continuing the work order.

The function determines:

- When to stop a work order.
- When to roll back a work order once it has been stopped. Once the function has been received, the SARM applies it as an asynchronous event to the specified work order.
- When to send a return status variable to indicate whether or not the operation was successful.
- A work order for which an aims_stop_wo function was received can go through two states:
  - WO_STOP_WAIT if rollback is required and is in progress. While in this state, the work order can be cancelled.
  - WO_STOPPED if rollback is not required or has finished. While in this state, the work order can be cancelled or its status changed to WO_HELD, WO_INIT, or WO_REVIEW.

A work order is stopped only if it is in the WO_IN PROGRESS state when the **aims_stop_wo** function is received. A request for a work order in any other state is rejected immediately without affecting the work order.

Table 4-109 aims_stop_wo Arguments

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	The work order identifier.	Yes	I
rollback	An integer value that specifies whether or not to roll back the work order before it is stopped. Valid values are: • 1 – Rollback • 0 – Do not rollback Any other value causes the work order to be stopped without rollback.	Yes	I

Table 4-109 (Cont.) aims_stop_wo Arguments

Name	Description	Req'd	(I)nput/ (O)utput
ret_status	A return parameter that stores the return value of the RPC. Valid values are: • 0 – Request to stop the work order was not accepted. • 1 – Request to stop the work order was accepted.	Yes	O
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
userid	Optional user identification for audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I

Localizing International Messages

Localization is the process of preparing a product for use with a single language and character set. Localization can include:

- Translating the user interface and documentation
- Adapting time, date, and number formats
- Adding punctuation conventions
- Reconstructing icons and symbols

With the support of the ASAP localization toolkit, you can localize software and non-software components to any language based on the Roman alphabet (English, German, French, Spanish, etc.). Localization usually involves translating the user interface and documentation and adapting time, date, and number formats. In some cases, more significant changes may be required, and sometimes icons, symbols, metaphors, and even concepts must be reconsidered.

The localization toolkit does not let you localize or translate the system messages generated by third-party tools or operating systems.

The stored procedures locate international messages in the SARM database. The default language of American English is provided in the base release for ASAP. You can use it as a guide for defining other languages in ASAP.

Such messages are generated by the SARM and logged in the SARM database. They may be retrieved by the SRP and passed back to the originating system.

Use the following stored procedures to add, remove, and query international messages.

- SSP_new_intl_msg
- SSP_del_intl_msg
- SSP_list_intl_msg

Table 4-110 lists the current messages used by ASAP and distributed as part of the core release.

Table 4-110 ASAP Messages

lang_cd	msg_id	type	message	var_description
USA	1	D	Work Order %s Timed Out	WO Id: %s
USA	2	D	ASDL Command %s Skipped	ASDL: %s
USA	3	D	Cannot Find Mandatory Parameter %s, ASDL %s Fails	Parameter: %s, ASDL: %s
USA	4	D	Soft Error on ASDL %s, WO Processing Continuing	ASDL: %s
USA	5	D	ASDL %s of SRQ %d Completed	ASDL: %s, SRQ Id: %d
USA	6	D	ASDL %s of SRQ %d Failed	ASDL: %s, SRQ Id: %d
USA	7	D	Start of ASDL Provisioning Request for SRQ %d	SRQ Id: %d
USA	8	D	Sent ASDL %s to NE, Awaiting NE Response	ASDL: %s
USA	9	D	Unable to get ASDL Command %s of SRQ %d	ASDL: %s, SRQ Id: %d
USA	10	D	SRQ %d (Last CSDL %s) has Completed	SRQ Id: %d, CSDL: %s
USA	11	D	CSDL %s of SRQ %d has Completed	CSDL: %s, SRQ Id: %d
USA	12	D	Will Retry ASDL Command %s of SRQ %d. Current Retry # is %d	ASDL: %s, SRQ Id: %d, Retry #: %d
USA	13	D	ASDL Command %s of SRQ %d Failed after %d Retries	ASDL: %s, SRQ Id: %d
USA	14	D	NE %s Unavailable while Processing %s	Host Clli: %s, ASDL: %s
USA	15	D	Q Info: Queued: %02d:%02d:%02d, Start: %02d:%02d:%02d, Comp: %02d:%02d:%02d	NEP Queue Information
USA	16	D	ASDL Failure Msg: %s	NE %s ASDL Failed Message
USA	17	D	ASDL Command %s of SRQ %d Failed on Unknown NE Return Status	ASDL: %s, SRQ Id: %d
USA	18	D	NE Command: %s\nASDL Command: %s	NE Command Returned From NE: %s
USA	19	D	Network Element Routing Error, Failed SRQ %d	SRQ Id: %d
USA	20	D	ASDL %s of SRQ %d Rollback Ignored	ASDL: %s, SRQ Id: %d
USA	21	D	NE %s Unavailable while Rolling Back ASDL %s of SRQ %d	Host Clli: %s, ASDL: %s, SRQ Id: %d
USA	22	D	Roll Back ASDL %s Sent to NE	ASDL: %s

Table 4-110 (Cont.) ASAP Messages

lang_cd	msg_id	type	message	var_description
USA	23	D	Roll Back ASDL %s Rejected by NE %s	ASDL: %s
USA	24	D	Will Retry Roll Back of ASDL %s, SRQ %d, Current Retry # %d	ASDL: %s, SRQ Id: %d, Retry #: %d
USA	25	D	Roll Back ASDL %s, SRQ %d Failed After %d Retries	ASDL: %s, SRQ Id: %d, # Retries: %d
USA	26	D	Roll Back ASDL %s, SRQ %d Completed	ASDL: %s, SRQ Id: %d
USA	27	D	Roll Back ASDL %s of SRQ %d Failed\nNEP Message %s	ASDL: %s, SRQ Id: %d, NEP Message: %s
USA	28	D	Roll Back ASDL %s Failed due to Unknown NE Return Status	ASDL: %s
USA	29	D	State Table Syntax Error Processing %s	ASDL: %s
USA	30	D	Error Detected Loading State Table for ASDL %s	ASDL: %s
USA	31	D	Unknown ASDL Error %s	ASDL: %s
USA	32	D	Continue to Process Next ASDL	ASDL: %s
USA	33	D	N.E. Host %s	ASDL: %s Host %s
USA	34	S	Invalid ASDL Parameter Type in Configuration	-
USA	35	D	No ASDL Configuration defined for %s	ASDL: %s
USA	36	D	Network Element %s is in maintenance	Host: %s
USA	37	D	Port Failure on Connection to %s	Host: %s
USA	38	S	SRQ Provisioning Stopped	
USA	39	D	Updated for ASDL %s of SRQ %d	ASDL: %s SRQ %d
USA	40	D	End of Indexed Parameters for ASDL %s	ASDL: %s
USA	41	D	ASDL %s Provisioning Request to %s	ASDL: %s Host: %s
USA	42	D	ASDL %s for %s Route to NE %s	ASDL: %s MCLI/DN: %s Host: %s
USA	43	D	Delay failure threshold exceeded, SRQ Provisioning Stopped	-
USA	44	S	Delay failure with rollback required, SRQ Provisioning Stopped	-

SARM provisioning interface

This section covers the functions for the SARM configuration. The following topics are discussed:

- SARM Interface RPCs
- Update RPC Interface Definitions
- Control Interface RPCs
- Retransmission of Recent Change Messages
- Real-Time Performance Data Gathering
- Switch Activation and Deactivation

SARM interface RPCs

This section defines the function interface that accesses the SARM database. These procedures provide query and update facilities that allow an OCA or JSRP application to monitor system performance, perform error correcting, and resubmit failed work orders.

This subsection lists the syntax, descriptions, parameters, and results for the functions that apply to the Inquiry RPC interface definition.

SAS_asdl_counts

This function generates a list of statistical information for an ASDL command on the service request (specified by srq_id and asdl_unid).

For more information about using functions, see "Oracle Execution Examples."

Table 4-111 SAS_asdl_counts Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier.	Yes	I
asdl_unid	ASDL identifier.	Yes	I
num_sw_history	Number of ASDL switch history occurrences for this ASDL.	Yes	O
num_params	Number of parameters used by this ASDL.	Yes	1

SAS_asdl_list

This function retrieves a list of ASDL commands and their information for a CSDL on the service request (specified by srq_id and csdl_seq_no). This information is retrieved from tbl_asdl_log.

For more information about using functions, see "Oracle Execution Examples."

Table 4-112 SAS_asdl_list Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I

Table 4-112 (Cont.) SAS_asdl_list Parameters

Name	Description	Req'd	(I)nput/ (O)utput
csdl_seq_no	CSDL sequence number.	Yes	I

Table 4-113 SAS_asdl_list Results

Name	Datatype	Description
asdl_cmd	TYP_asdl_cmd	ASDL command.
asdl_stat	TYP_asdl_stat	ASDL status updated while this ASDL is in progress.
asdl_unid	TYP_seq_no	A unique ASDL identifier generated when an ASDL is routed to an NE.
host_clli	TYP_clli	The Host NE to which the ASDL command is routed by the SARM.
rollback_asdl	TYP_asdl_cmd	ASDL command used to roll back the original ASDL.
comp_dts	datetime	The completion date and time of the ASDL processing.
rollback_dts	datetime	The date and time of the ASDL rollback, if rollback was required on this ASDL.
description	varchar(40)	ASDL command description.
queue_dts	datetime	Queuing date and time for the ASDL command to the NE.
start_dts	datetime	Provisioning start date and time for the ASDL command.
retry_count	TYP_long	Number of times the ASDL command was retried.
stat_text	TYP_stat_text	The status text for the ASDL.

SAS_asdl_parms

This function retrieves the ASDL parameters for an ASDL command on the service request that has been provisioned. These parameters are used and generated during the provisioning process. This information is retrieved from tbl_srq_asdl_parm.

For more information about using functions, see "Oracle Execution Examples."

Table 4-114 SAS_asdl_parms Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I
asdl_unid	ASDL identifier.	Yes	I

Table 4-115 SAS_asdl_parms Results

Name	Datatype	Description
parm_lbl	TYP_parm_lbl	Parameter name.

Table 4-115 (Cont.) SAS_asdl_parms Results

Name	Datatype	Description
parm_vlu	TYP_parm_vlu	Parameter value.

SAS_asdl_sw_history

This function retrieves the switch history for an ASDL command from tbl_srq_log on the service request (specified by srq_id and asdl_unid).

For more information about using functions, see "Oracle Execution Examples."

Table 4-116 SAS_asdl_sw_history Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I
asdl_unid	ASDL identifier.	Yes	I

Table 4-117 SAS_asdl_sw_history Results

Name	Datatype	Description
evt_dt_tm	datetime	Event timestamp.
csdl_seq_no	TYP_seq_no	Sequence number for the CSDL associated with the ASDL command.
srq_stat	TYP_srq_stat	Service request status.
evt_text	TYP_evt_text	Switch history record. Note that each returned event text field may contain new line characters within the event text itself.

SAS_csdl_counts

This function generates statistical information for a CSDL command on the service request.

Affected tables:

- tbl_srq_parm

For more information about using functions, see "Oracle Execution Examples."

Table 4-118 SAS_csdl_counts Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier.	Yes	I
csdl_seq_no	CSDL sequence number.	Yes	I

Table 4-118 (Cont.) SAS_csdl_counts Parameters

Name	Description	Req'd	(I)nput/ (O)utput
num_events	Number of CSDL provisioning events for this CSDL.	Yes	O
num_sw_history	Number of ASDL switch history occurrences for this ASDL.	Yes	O
num_asdl	Number of ASDLs which have been executed for this CSDL.	Yes	O
num_csdl_params	Number of CSDL parameters for this CSDL.	Yes	O

SAS_csdl_event_history

This function generates a listing of provisioning events for a CSDL command on the service request. This information is contained in tbl_srq_log, tbl_stat_text.

For more information about using functions, see "Oracle Execution Examples."

Table 4-119 SAS_csdl_event_history Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I
csdl_seq_no	CSDL sequence number.	Yes	I

Table 4-120 SAS_csdl_event_history Results

Name	Datatype	Description
evt_dt_tm	datetime	Date and time that the event occurred.
evt_text	TYP_evt_text	Switch history record. Note that each returned event text field may contain new line characters within the event text itself.
srq_evt	TYP_srq_evt	The service request log event.
srq_stat	TYP_srq_stat	Status of the service request at the time the event occurred.
stat_text	TYP_stat_text	Status text for the service request.

SAS_csdl_list

This function generates a list of CSDL commands and information pertaining to them for a service request. This information is contained in tbl_srq_csdl, tbl_stat_text.

For more information about using functions, see "Oracle Execution Examples."

Table 4-121 SAS_csdl_list Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I

Table 4-122 SAS_csdl_list Results

Name	Datatype	Description
csdl_seq_no	TYP_seq_no	Sequence number for the CSDL associated with the ASDL command.
actn_noun_lbl	TYP_csdl_cmd	CSDL command.
csdl_st	TYP_csdl_st	CSDL command status.
asdl_seq_no	TYP_seq_no	ASDL command sequence number.
index_parm_cnt	TYP_seq_no	Index parameter count for the current ASDL command.
asdl_route	TYP_asdl_route	Routing status of the current ASDL command.
csdl_id	TYP_unid	CSDL identification specified by SRP.
asdl_route_rep_1	TYP_asdl_route	Routing status of the current ASDL command.
csdl_type	TYP_csdl_type	Type of CSDL command (for example, ORIGINAL, REVISION, SAS_ORIGINAL, SAS_REVISION).
orig_seq_no	TYP_seq_no	Original CSDL command sequence number associated with revision CSDL commands.
estimate	TYP_long	Initial estimate for ASDL processing calculated when the CSDL command was received.
start_dts	datetime	Provisioning start date and time for the first ASDL command of the CSDL command.
abort_dts	datetime	Abort time for the CSDL command.
failure_dts	datetime	Failure time for the CSDL command.
comp_dts	datetime	Completion time for the CSDL command.
update_dts	datetime	Last update timestamp for the CSDL command.
update_uid	TYP_user_id	User ID of user who last updates the CSDL command.
prov_sequence	TYP_seq_no	Provisioning sequence number for the CSDL command.
stat_text	TYP_stat_text	CSDL status text.
description	varchar(40)	CSDL description.

SAS_csdl_parms

This function retrieves the CSDL parameters for a CSDL command on the service request (specified by srq_id and csdl_seq_no). This information is retrieved from tbl_srq_parm.

For more information about using functions, see "Oracle Execution Examples."

Table 4-123 SAS_csdl_parms Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I
csdl_seq_no	CSDL sequence number.	Yes	I

Table 4-124 SAS_csdl_parms Results

Name	Datatype	Description
parm_lbl	TYP_parm_lbl	Parameter name.
parm_vlu	TYP_parm_vlu	Parameter value.

SAS_csdl_sw_history

This function retrieves the switch history for a CSDL command on the service request (specified by srq_id and csdl_seq_no) from tbl_srq_log.

For more information about using functions, see "Oracle Execution Examples."

Table 4-125 SAS_csdl_sw_history Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I
csdl_seq_no	CSDL sequence number.	Yes	I

Table 4-126 SAS_csdl_sw_history Results

Name	Datatype	Description
evt_dt_tm	datetime	Event timestamp.
csdl_seq_no	TYP_seq_no	Sequence number for the CSDL associated with the ASDL command.
srq_stat	TYP_srq_stat	Service request status.
evt_text	TYP_evt_text	Switch history record. Note that each returned event text field may contain new line characters within the event text itself.

SAS_info_parms

This function retrieves the information parameters for a work order (specified by wo_id). This information is retrieved from tbl_info_parm.

For more information about using functions, see "Oracle Execution Examples."

Table 4-127 SAS_info_parms Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
wo_id	Work order identifier.	Yes	I

Table 4-128 SAS_info_parms Results

Name	Datatype	Description
parm_lbl	TYP_parm_lbl	Parameter name.
parm_group	TYP_parm_grp	Parameter group.
parm_vlu	TYP_parm_vlu	Parameter value.
csdl_seq_no	TYP_seq_no	Sequence number for the CSDL which generated the information parameter.
csdl_cmd	TYP_csdl_cmd	CSDL command name.
csdl_id	TYP_unid	CSDL identification specified by SRP.
description	archar(40)	CSDL description.

SAS_map_srq_id

This function maps the service request ID to the work order identifier.

For more information about using functions, see "Oracle Execution Examples."

Table 4-129 SAS_map_srq_id Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier.	Yes	I
wo_id	Work order identifier associated with the service request. Set to NULL if srq_id is invalid.	Yes	O

SAS_map_wo_id

This function maps the work order identifier to its service request ID. This information is stored in tbl_srq.

For more information about using functions, see "Oracle Execution Examples."

Table 4-130 SAS_map_wo_id Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier.	Yes	I

Table 4-130 (Cont.) SAS_map_wo_id Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier associated with the work order. Set to NULL if the work order is not found.	Yes	I

SAS_wo_detail

This function retrieves the detailed information for a work order specified in the input parameter wo_id. This information is retrieved from tbl_wrk_ord.

For more information about using functions, see "Oracle Execution Examples."

Table 4-131 SAS_wo_detail Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
wo_id	Work order identifier.	Yes	I

Table 4-132 SAS_wo_detail Results

Name	Datatype	Description
wo_id	TYP_wo_id	Work order identifier associated with the ASDL command.
wo_stat	TYP_wo_stat	Work order status.
org_unit	TYP_org_unit	Organization unit specified on order.
sched_dts	datetime	Due date and time.
orig_login	TYP_user_id	Originator of the work order in the host system.
comp_dts	datetime	The completion date and time of the ASDL processing.
srp_id	TYP_code	Name of the SRP that transmits the work order.
update_dts	datetime	Update date and time.
revs_flag	char(1)	Indicates whether any revisions were made.
exceptions	char(1)	Indicates whether there are any exceptions in the completion of the work order.
pend_cancel	char(1)	Indicates whether there is a pending order cancellation for this work order.
rollback_stat	TYP_status	Work order roll back status.
command	TYP_long	Provisioning command transmitted from SRP
srq_id	TYP_srq_id	Service request identifier.
grp_cd	TYP_grp_cd	Action to be taken by ASAP, which is transmitted from SRP.
srq_pri	TYP_srq_pri	Work order priority.

Table 4-132 (Cont.) SAS_wo_detail Results

Name	Datatype	Description
proc_type	TYP_proc_typ	Process type which specifies if the work order is immediate or a future type.

SAS_wo_by_host_clli

This function retrieves a list of work orders by host_clli in combination with other parameters from tbl_wrk_ord and tbl_srq.

For more information about using functions, see "Oracle Execution Examples."

Table 4-133 SAS_wo_by_host_clli Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	Host CLLI identifier.	No	I
sched_dts	The date and time the work order is scheduled to begin provisioning.	No	I
wo_stat	Work order status.	No	I
wo_id	Work order identifier or partial work order identifier. You can use wildcards.	No	I

SAS_wo_list

This function retrieves a list of work orders based on specified query criteria from tbl_wrk_ord, tbl_stat_text.

Three independent types of queries are supported by this function:

- By specifying wo_id, the work orders exactly matching the work order ID are selected.
- If wo_id is not specified (set to null), then the wo_stat, sched_dts_from, sched_dts_to and org_unit can be specified individually or in any combination. Work orders are retrieved based on all the parameters specified (and boolean relationships are used among the parameters).
- When no parameters are supplied (that is, all fields are NULL), all work orders in SARM are retrieved.

For more information about using functions, see "Oracle Execution Examples."

Table 4-134 SAS_wo_list Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
wo_state	Work order status.	No	I
sched_dts_from	Start point of due date range.	No	I

Table 4-134 (Cont.) SAS_wo_list Parameters

Name	Description	Req'd	(I)nput/ (O)utput
sched_dts_to	End point of due date range.	No	I
org_unit	Organization Unit Code. This could be a partial org_unit with wildcards.	No	I
wo_id	Work order identifier or partial work order identifier. You can use wildcards.	No	I

Table 4-135 SAS_wo_list Results

Name	Datatype	Description
wo_id	TYP_wo_id	Work order identifier associated with the ASDL command.
srp_id	TYP_code	The logical SRP name.
wo_stat	TYP_wo_stat	Work order status.
org_unit	TYP_org_unit	Organization unit specified on order.
sched_dts	datetime	Due date and time.
orig_login	TYP_user_id	Originator of the work order in the host system.
lock_uid	TYP_user_id	User ID of the user who has locked the order for update.
lock_dts	datetime	Lock timestamp.
stat_text	TYP_stat_text	The work order status.

SAS_wo_parms

This function retrieves the global parameters for a service request (specified by srq_id) from tbl_srq_parm.

For more information about using functions, see "Oracle Execution Examples."

Table 4-136 SAS_wo_parms Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier.	Yes	I

Table 4-137 SAS_wo_parms Results

Name	Datatype	Description
parm_lbl	TYP_parm_lbl	Parameter name.
parm_vlu	TYP_parm_vlu	Parameter value.

Update RPC interface definitions

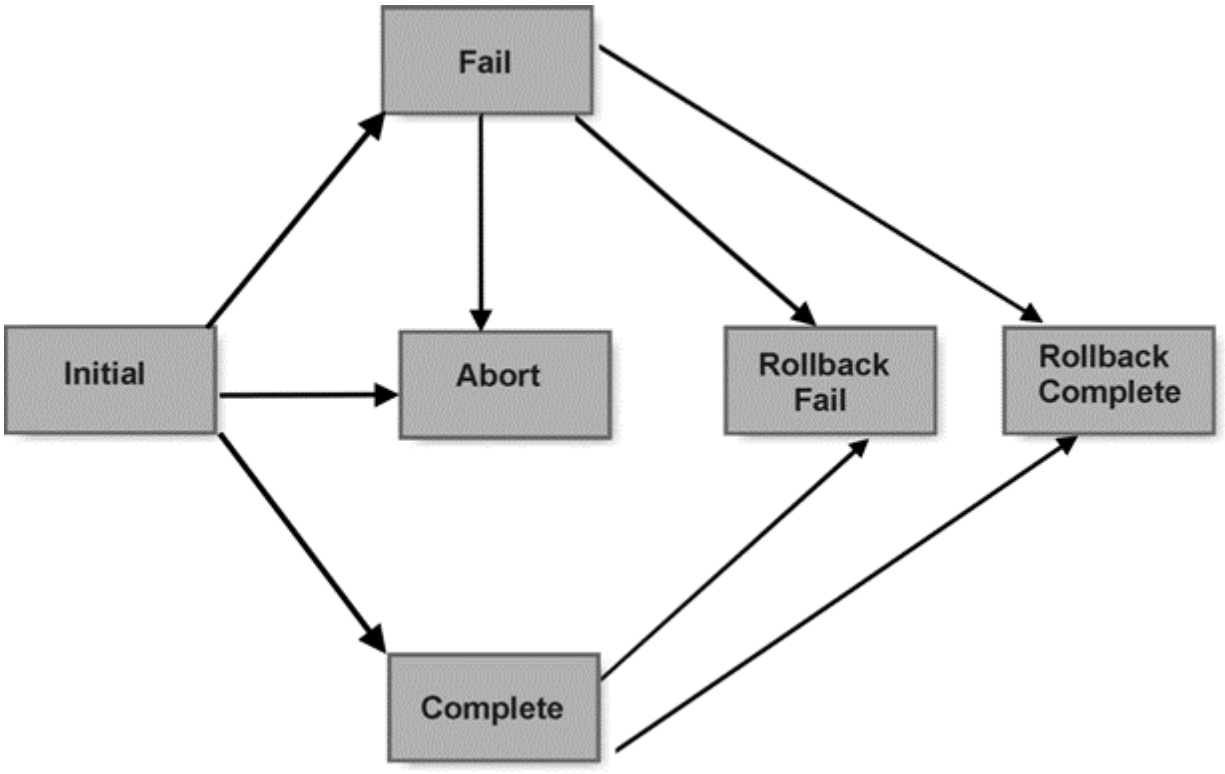
This section describes the RPC interface definitions.

CSDL processing model

> **Note:**
>
> The work order must be in the LOCK state before the update can be performed. To avoid race conditions when updating a work order, the order must be locked using SAS_lock_wo.

Once the work order has been locked for processing, the following CSDL state transition model must be observed.

Figure 4-1 CSDL State Transition Model



Order management transactions:

The following table contains work order transactions and their associated functions.

Table 4-138 Order management transactions:

Transaction	function	Description
Change Due Date	SAS_change_due_dt	Changes the due date on one work order or a range of work orders.
Change Priority	SAS_change_priority	Changes the priority on the locked work order.
Lock service request	SAS_lock_wo	Locks the work order for update.
Hold service request	SAS_hold_wo	Holds the work order in ASAP to prevent provisioning. If the order is to be reviewed, the review indicator should be set.
Release service request to pending queue	SAS_release_wo	Releases a work order from the HELD, REVIEW, or LOCKED state to the ready queue for provisioning. Before you can update a CSDL command on a locked order, the CSDL command must be in the INITIAL state. This is done by using the SAS_release_wo procedure.
Global parameter maintenance	1. SAS_lock_wo 2. SAS_add_wo_parm 3. SAS_delete_wo_parm 4. SAS_updt_wo_parm 5. SAS_release_wo	1. Locks the work order for update. 2. Creates a global work order parameter for the work order. 3. Deletes an existing work order parameter on the work order. 4. Updates an existing work order parameter on the work order. 5. Releases the work order to the pending, held, or review queue.

CSDL Management Transactions:

Table 4-139 CSDL Management Transactions

Transaction	Function	Description
Add a new CSDL command to an order	1. SAS_lock_wo 2. SAS_csdl_list (Optional) 3. SAS_renumber_csdl (Optional) 4. SAS_add_csdl 5. SAS_add_csdl_parm 6. SAS_release_wo	1. Locks the work order for update. 2. Determines the sequence insertion point for the CSDL command within the order. 3. Optional CSDL sequencing renumber that creates an insertion point for the new CSDL command. 4. Adds the CSDL command to the order for provisioning. 5. Adds the necessary parameters to the CSDL command. 6. Releases the work order to the pending, held, or review queue.

Table 4-139 (Cont.) CSDL Management Transactions

Transaction	Function	Description
Delete a CSDL command from the order	1. SAS_lock_wo 2. SAS_csdl_list (Optional) 3. SAS_abort_csdl 4. SAS_release_wo	1. Locks the work order for update. 2. Selects an INITIAL state CSDL command on the order. 3. Aborts the CSDL command to prevent provisioning. Do not set the “copy CSDL" parameter. 4. Releases the work order to the pending, held, or review queue.
Update a CSDL command on the order	-	Aborts the specified CSDL command and then creates a new CSDL command.
CSDL parameter maintenance	1. SAS_lock_wo 2. SAS_csdl_list (Optional) 3. SAS_add_csdl_parm 4. SAS_csdl_parms (Optional) 5. SAS_updt_csdl_parm 6. SAS_delete_csdl_parm 7. SAS_release_wo	1. Locks the work order for update. 2. Selects an INITIAL state CSDL command on the order. 3. Creates a new CSDL parameter on a specific CSDL command. 4. Selects the CSDL parameter to be modified or deleted. 5. Updates the CSDL parameter value. 6. Deletes the CSDL parameter. 7. Releases the work order to the pending, held, or review queue.
Change the CSDL command provisioning sequence	1. SAS_lock_wo 2. SAS_csdl_list (Optional) 3. SAS_renumber_csdl (Optional) 4. SAS_move_csdl 5. SAS_release_wo	1. Locks the work order for update. 2. Selects an INITIAL state CSDL command on the order. 3. Optional CSDL sequencing renumber that creates an insertion point for the CSDL command. 4 .Moves the selected CSDL command within the order. 5. Releases the work order to the pending, held, or review queue.

Failed Order Processing Transactions:

Table 4-140 Failed Order Processing Transactions:

Transaction	Function	Description
Fix an order with inconsistent or missing parameters (Note: This includes routing errors.)	1. SAS_wo_list (Optional) 2. SAS_lock_wo 3. SAS_csdl_list (Optional) 4. SAS_abort_csdl 5. SAS_release_wo, SAS_hold_wo, SAS_resubmit_wo	1. Selects the failed orders or a specific order. 2. Locks the work order for update. 3. Analyzes the CSDL commands on the order to determine the failed CSDL. Fixes the global work order parameters as described in “Global parameter maintenance". 4. Aborts the failed CSDL command and creates a revision copy of the CSDL command. Fixes the CSDL parameters as described in “CSDL parameter maintenance". 5. Releases the work order to the pending, held, or review queue or resubmits for immediate processing.
Retry a failed order with no changes	1. SAS_wo_list (Optional) 2. SAS_lock_wo 3. SAS_csdl_list (Optional) 4. SAS_release_wo, SAS_resubmit_wo	1. Selects the failed orders or a specific order. 2. Locks the work order for update. 3. Analyzes the CSDL commands on the order to determine the failed CSDL. 4. Releases the work order to the pending queue or resubmits for immediate processing.
Fix a failed order with a translation error	1. SAS_lock_wo 2. SAS_release_wo	1. Locks the work order for update.Fixes the global and/or CSDL parameters that caused the translation error. 2. Releases the work order to the pending, held, or review queue.

functions

This section lists the syntax, descriptions, parameters, and results for the functions that apply to the Update RPC interface definition.

SAS_abort_csdl

This function aborts a CSDL and creates a copy of that CSDL. When the copy is created, the copy can be edited, or changes can be made to the CSDL parameters to fix provisioning problems. Upon the order completion, the original CSDL and copy of the CSDL command is available for revisions and for further processing. Updates tbl_srq_parm.

For more information about using functions, see "Oracle Execution Examples."

Table 4-141 SAS_abort_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the CSDL command within the service request.	Yes	I

Table 4-141 (Cont.) SAS_abort_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
copy_flag	Yes/no flag indicating whether or not a copy of the CSDL command should be made.	Yes	I
new_seq_no	Sequence number used to identify the copy of the CSDL command within the service request. If the copy cannot be created, this parameter is returned as -1.	Yes	O
userid	Optional user identification for audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_abort_wo

This function aborts a work order that has not begun provisioning, or one that has failed during the translation or provisioning phase. Affected tables: tbl_wrk_ord, tbl_srq,

For more information about using functions, see "Oracle Execution Examples."

Table 4-142 SAS_abort_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier or partial work order identifier. For a partial work order, use a syntax similar to “C1234%". Using partial order numbers like “%123" when using this function requires considerable system resources and therefore may affect system performance.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_add_csdl

This function adds a CSDL command to the service request (tbl_srq_csdl).

For more information about using functions, see "Oracle Execution Examples."

Table 4-143 SAS_add_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the sequence of the CSDL command within the service request.	Yes	O
csdl_cmd	The CSDL command to add to the service request.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
rev_seq_no	Sequence number for the original CSDL command used to identify that this is a revision.	Yes	I
csdl_id	CSDL identification for the SRP.	Yes	I
sequence	Sequence number.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_add_csdl_parm

This function adds CSDL parameters to an existing CSDL command, which is in the INIT state (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-144 SAS_add_csdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the CSDL command within the service request.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I
parm_vlu	CSDL parameter value.	Yes	I

Table 4-144 (Cont.) SAS_add_csdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_add_wo_parm

This function adds global work order parameters to an existing service request (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-145 SAS_add_wo_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I
parm_vlu	CSDL parameter value.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_change_due_dt

This function changes the due date on specified work orders in tbl_srq. It is not necessary for work orders to be in the LOCKED state. The criteria must comply with the following rules:

- Either wo_id or wo_stat must be supplied.
- wo_id can be supplied with the wildcard % for an approximate search.
- If wo_id is given, wo_stat, date range and org_unit are ignored.
- If wo_id is not given, but wo_stat is given, date range and org_unit take effect.

For more information about using functions, see "Oracle Execution Examples."

Table 4-146 SAS_change_due_dt Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier or a partial work order identifier. For a partial work order use a syntax similar to “C1234%". Note: using partial order numbers like “%123" when using this function requires considerable system resources and therefore may affect system performance.	No	I
due_dt	New due date for the work orders.	Yes	I
wo_stat	Work order status. This parameter cannot be set if the wo_id parameter is specified.	No	I
from_dt, to_dt	Optional due date range for specifying the work orders to update. These parameters cannot be set if the wo_id parameter is specified.	No	I
org_unit	Optional organization unit identifier.	No	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_change_priority

Changes the priority on the work orders specified by the query criteria. Work orders do not need to be in the LOCKED state. Affected tables: tbl_wrk_ord, tbl_srq. The query criteria must comply with the following rules:

- Either wo_id or wo_stat must be supplied.
- wo_id can be supplied with the wildcard % for an approximate search (like in SQL).
- If wo_id is given, wo_stat, date range and org_unit are ignored.
- If wo_id is not given, but wo_stat is given, date range and org_unit take effect.

For more information about using functions, see "Oracle Execution Examples."

Table 4-147 SAS_change_priority Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier or partial work order identifier. For a partial work order use syntax similar to “C1234%". Note: using a partial order numbers like “%123" when using this function requires considerable system resources and therefore may affect system performance.	No	I
priority	New priority for the work order(s).	Yes	I
wo_stat	Optional work order status. This parameter cannot be set if the wo_id parameter is specified.	No	I
from_dt, to_dt	Optional due date range to specify the work orders to update. These parameters cannot be set if the wo_id parameter is specified.	No	I
org_unit	Optional organization unit identifier.	No	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_delete_csdl_parm

This function deletes CSDL parameters from an existing CSDL command (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-148 SAS_delete_csdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the CSDL command within the service request.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I

Table 4-148 (Cont.) SAS_delete_csdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_delete_wo_parm

This function deletes work order parameters from a service request (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-149 SAS_delete_wo_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
ora_option	If option is set to Y, the parameters deleted are based on a wildcard search; if option is N, only the parameter exactly matching parm_lbl will be deleted.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_get_csdl_stat

This function is used to retrieve the status of a CSDL command in a service request.

For more information about using functions, see "Oracle Execution Examples."

Table 4-150 SAS_get_csdl_stat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	The service request identifier.	Yes	I
csdl_seq_no	The sequence number used to identify the CSDL command within the service request.	Yes	I
csdl_stat	The status of the CSDL when the function is returned.	Yes	O

SAS_get_srq_stat

This function is used to retrieve the status of a service request from tbl_srq.

For more information about using functions, see "Oracle Execution Examples."

Table 4-151 SAS_get_srq_stat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	The service request identifier.	Yes	I
srq_stat	The status of the service request when the function is returned.	Yes	O

SAS_get_wo_stat

This function is used to retrieve the status of a work order from tbl_wrk_ord.

For more information about using functions, see "Oracle Execution Examples."

Table 4-152 SAS_get_wo_stat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	The work order identifier.	Yes	I
wo_stat	The status of the work order when the function is returned.	Yes	O

SAS_hold_wo

This function holds a work order specified by wo_id. Only work orders in the LOCKED state can be held. Work order status information is contained in tbl_wrk_ord.

For more information about using functions, see "Oracle Execution Examples."

Table 4-153 SAS_hold_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier.	Yes	I
review	Yes/no flag indicating whether or not the order should be set for REVIEW.	No	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_lock_wo

This function locks the work order specified and sets its state to LOCKED. Only work orders in INIT, REVIEW, TRANSLATION FAILED, HELD and FAILED state can be locked. Transfer_WO and latency_timout parameters can be used in high availability mode only. Work order status information is contained in tbl_wrk_ord.

For more information about using functions, see "Oracle Execution Examples."

Table 4-154 SAS_lock_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
result	Set to: • 0 – Indicates successful order lock • 1 – Indicates order lock failed	Yes	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

Table 4-154 (Cont.) SAS_lock_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
latency_timeout	Deprecated.		
transfer_wo	Deprecated.		

SAS_move_csdl

This function changes the provisioning sequence number for an existing CSDL command on a service request. This information is contained in tbl_srq_csdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-155 SAS_move_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
srq_id	Service request identifier used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the CSDL command within the service request.	Yes	I
prov_seq	The new provisioning sequence number for the CSDL on the service request.	Yes	I
userid	Optional user identification for audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_release_wo

This function releases a work order for ASAP processing. The work order must be in a HELD, REVIEW or LOCKED state before it can be released. Work order status information is contained in tbl_srq_csdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-156 SAS_release_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier.	Yes	I
immediate	A flag indicating if the order should be released immediately or on the due date. Possible values: • Y – Immediate release • N – Release on due date	No	I
due_dt	Optional due date used to specify a new due date for the work order. If the order is immediate, then this parameter is ignored.	No	I
rel_to_fail	Yes/no flag indicating the value of new_wo_stat. Possible values: • N – new_wo_stat is set to WO_INIT. • Y – new_wo_stat is set to WO_FAILED.	No	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_renumber_csdl

This function renumbers the provisioning sequence of all CSDL commands on a service request using the interval specified by the interval parameter. This information is contained in tbl_srq_csdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-157 SAS_renumber_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
interval	Optional renumbering interval for the CSDL commands within the order (defaults to 5).	No	I
userid	Optional user identification for audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I

Table 4-157 (Cont.) SAS_renumber_csdl Parameters

Name	Description	Req'd	(I)nput/ (O)utput
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_resubmit_wo

This function resubmits a failed or locked work order for provisioning. The work order must be in the FAILED or LOCKED state before it can be resubmitted. Work order information is contained in tbl_wrk_ord.

For more information about using functions, see "Oracle Execution Examples."

Table 4-158 SAS_resubmit_wo Parameters

Name	Description	Req'd	(I)nput/ (O)utput
wo_id	Work order identifier.	Yes	I
abort_cur_csdl	Yes/no flag indicating whether or not the current CSDL should be aborted before resubmitting the order (defaults to No).	No	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_updt_csdl_parm

This function updates the CSDL parameters of an existing CSDL command in a service request (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-159 SAS_updt_csdl_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
csdl_seq_no	Sequence number used to identify the CSDL command within the service request.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I
parm_vlu	CSDL parameter value.	Yes	I
userid	Optional user identification for audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S – SRQ log • W – Work order audit log • B – Both • N – Neither	No	I
update_dts	Date and time of the last update.	No	I

SAS_updt_wo_parm

This function updates global work order parameters on the service request (tbl_srq_parm).

For more information about using functions, see "Oracle Execution Examples."

Table 4-160 SAS_updt_wo_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
srq_id	Service request identifier that is used to specify the work order.	Yes	I
parm_lbl	CSDL parameter name.	Yes	I
parm_vlu	CSDL parameter value.	Yes	I
userid	Optional user identification for the audit log. Set to NULL to disable the audit log. Set to the user ID of the user who executes the procedure (for instance, guest) to enable the audit log.	No	I
evt_text	The text of the message associated with the event.	No	I
audit_flag	Indicates which audit log receives the message: • S for SRQ log • W for work order audit log • B for both • N for neither	No	I
update_dts	Date and time of the last update.	No	I

Control interface RPCs

This section describes the function interface that accesses the dynamic data in the Control database.

SAS_list_alarm_log

This function lists system-generated alarms contained in tbl_alarm_log. Alarms can be retrieved using the ID of the event that generates the alarm (event_unid), system alarm code (alarm_code), or alarm ID (alarm_unid), or any combination of these three parameters (and Boolean relationships). If no parameters are specified, all the alarms saved to the alarm log.

For more information about using functions, see "Oracle Execution Examples."

Table 4-161 SAS_list_alarm_log Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
event_unid	Unique ID of the event that generated the alarm.	No	I
alarm_code	The alarm code of the generated alarm.	No	I
alarm_unid	Unique ID of the alarm.	No	I

Table 4-162 SAS_list_alarm_log Results

Name	Datatype	Description
event_unid	TYP_unid	Unique ID of the event that generated the alarm.
alarm_code	TYP_code	The alarm code of the generated alarm.
alarm_unid	TYP_unid	Unique ID of the alarm.
start_dts	datetime	The start date and time of the system alarm.
escalation_dts	datetime	The date and time of the last alarm escalation.
clear_dts	datetime	The date and time when the alarm was cleared.

SAS_list_appl_proc

This function lists ASAP application configuration information contained in tbl_appl_proc. Application configuration information can be retrieved using the ASAP application server (appl_cd) or the ASAP startup sequence (start_seq), or by combining these two parameters. If neither parameter is specified, configuration information for all the ASAP application servers is retrieved.

For more information about using functions, see "Oracle Execution Examples."

Table 4-163 SAS_list_appl_proc Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
appl_cd	The logical ASAP application client/server code, for instance SARM, NEP01, etc.	No	I
start_seq	Specifies the ASAP startup sequence. This allows control to be exercised over the sequence in which applications are started.	No	I

Table 4-164 SAS_list_appl_proc Results

Name	Datatype	Description
start_seq	TYP_start_seq	Startup sequence number.
appl_type	TYP_appl_type	Application type, i.e., Client (“C") or Server (“S"), Master Control Server (“M"), Remote Slave Control Server (“R").
appl_cd	TYP_code	ASAP logical Client/Server name.
control_svr	TYP_code	The logical ASAP control server that spawns and monitors the application.
description	TYP_desc	Brief description of the ASAP application.
diag_file	TYP_unix_file	Diagnostic file name where all application diagnostic messages are output.
auto_start	TYP_yesno	Determines if the application is started automatically when ASAP starts.
program	varchar(40)	The name of the executable UNIX file that corresponds to the application.
diag_level	TYP_diag_level	Diagnostic level of the ASAP application.
isactive	isactive	Indicates whether the application is currently active.
last_start	datetime	The last date and time when the application was started.
last_halt	datetime	The last date and time when the application was terminated.
last_abnormal	datetime	The last date and time when the application was abnormally terminated.
svr_type	varchar(8)	This field defines the type of server.

SAS_list_event_log

This function lists system events generated by ASAP applications. This information is retrieved from tbl_event_log. Events can be retrieved using event ID (event_unid), ASAP application server (appl_cd) or date range that the event is generated (from_dt and to_dt), or any combination of these parameters (and boolean relationships). If no parameters are specified, the events for all the ASAP application servers are retrieved.

For more information about using functions, see "Oracle Execution Examples."

Table 4-165 SAS_list_event_log Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
event_unid	Unique ID of the event that generated the alarm.	No	I
appl_cd	Logical name of the ASAP application server.	No	I
from_dt	The start from date and time in the date range that the events are generated.	No	I
to_dt	End date and time in the date range that the events are generated.	No	I

Table 4-166 SAS_list_event_log Results

Name	Datatype	Description
appl_cd	TYP_code	Logical name of the ASAP application that generated the system event.
event_type	TYP_event	The event type that determines if the alarm is to be generated when the event occurs.
event_unid	TYP_unid	Unique ID of the event that generated the alarm.
source_file	TYP_unix_file	Source file that corresponds to the ASAP application that generated the event.
source_line	TYP_short	Line number in the source file.
reason	TYP_reason	Brief description of the reason for the event.
evt_dts	datetime	Date and time of the system event.

SAS_list_proc_info

This function retrieves process information from tbl_process_info for a specified application server.

Table 4-167 SAS_list_proc_info Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
appl_cd	Logical name of the ASAP application server.	No	I
from_dt	The start from date and time in the date range that the application process information is logged.	No	I
to_dt	End date and time in the date range that the application process information is logged.	No	I

Table 4-168 SAS_list_proc_info Results

Name	Datatype	Description
appl_cd	TYP_code	ASAP logical Client/Server name.
info_dts	datetime	Date and time when process information was logged.
sys_events	TYP_short	The number of system events generated by the ASAP application process.
user_cpu	TYP_long	The user CPU usage of the process.
system_cpu	TYP_long	The system CPU usage of the process.

Real-time performance data gathering

To support the use of real-time performance monitoring tools, the SARM server maintains statistical data such as the number of orders that have been processed, flowed through, required manual intervention, and so on.

First, an Administration Server (ADMS) off-loads the performance data inquiry processing required by the SARM. Then, ADMS queries the SARM based on a system-configured parameter and responds to queries from the monitoring clients for real-time data. The polling requests from the clients are independent of the polling requests performed by the ADMS to the SARM. Finally, the ADMS updates the performance monitoring database periodically to generate historical information.

The RPCs defined in the Interface Definition are sent to ADMS for real-time information (ADM_*) and to the database server (PSP_*) for historical information.

For more information about using functions, see "Oracle Execution Examples."

Table 4-169 Real-time Performance Data Gathering Configuration Parameters

Name	Default Value	Config File	Description
POLL_TIMER_ORDER	120	Global	Polling time, in minutes, to query order-related statistics from the SARM.
POLL_TIMER_CSDL	120	Global	Polling time, in minutes, to query CSDL-related statistics from the SARM.
POLL_TIMER_ASDL	120	Global	Polling time, in minutes, to query ASDL related statistics from the SARM.
POLL_TIMER_NE	120	Global	Polling time, in minutes, to query NE-related statistics from the SARM.
POLL_TIMER_NE_ASDL	120	Global	The polling timer is limited to the following values: 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60, and 120. ADMS queries the SARM at these intervals to store historical information in the database. These queries occur on the exact hour and at regular intervals within the hour.

ADM_asdl_stats, PSP_asdl_stats

These RPCs query the ASDL command statistics. ADM_asdl_stats performs a wildcard match on the ASDL. For example, if the ASDL passed is “A-ADD_", statistics are returned for all ASDLs that start with “A-ADD_".

Similarly, the PSP RPC is used for the same ASDL, together with a timestamp range. Historical statistics are returned from tbl_perf_asdl for all ASDLs meeting the criteria.

For more information about using functions, see "Oracle Execution Examples."

Table 4-170 ADM_asdl_stats, PSP_asdl_stats Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
sarm	SARM where data originated.	No	I/O
asdl_cmd	ASDL command.	No	I
from_dts, to_dts	Historical inquiry timestamp range.	No	I

Table 4-171 ADM_asdl_stats, PSP_asdl_stats Results

Name	Datatype	Description
sarm	char(8)	SARM where data originated.
update_dts	datetime	Update timestamp.
asdl_cmd	TYP_asdl_cmd	ASDL command.
asdl_exe	int	Number of times an ASDL command was sent to the NEP.
asdl_fail	int	Number of times an ASDL command failed.
asdl_comp	int	Number of times an ASDL command successfully completed.
asdl_parm_avg	float	Average number of ASDL parameters on the ASDL command.
asdl_parm_min	int	Minimum number of parameters on the ASDL command.
asdl_parm_max	int	Maximum number of parameters on the ASDL command.
asdl_rbacks	int	Number of times the ASDL command was rolled back.
asdl_soft_err	int	Number of times the ASDL command returned with a soft error.
asdl_retries	int	Number of times the ASDL was executed.
asdl_skipped	int	Number of times that the ASDL command was skipped.
comp_time_avg	float	Average completion time of the ASDL command.
comp_time_min	float	Minimum completion time of the ASDL command.

Table 4-171 (Cont.) ADM_asdl_stats, PSP_asdl_stats Results

Name	Datatype	Description
comp_time_max	float	Maximum completion time of the ASDL command.

ADM_csdl_stats, PSP_csdl_stats

These RPCs query CSDL command statistics from tbl_perf_csdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-172 ADM_csdl_stats, PSP_csdl_stats Parameters

Name	Description	Req'd	(I)nput/(O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
sarm	SARM where data originated.	No	I/O
csdl_cmd	The CSDL command.	No	I
from_dts, to_dts	Historical inquiry timestamp range.	No	I

Table 4-173 ADM_csdl_stats, PSP_csdl_stats Results

Name	Datatype	Description
sarm	char(8)	SARM where data originated.
update_dts	datetime	Update timestamp.
csdl_cmd	csdl_cmd	CSDL command name.
csdl_rcvd	int	Number of times CSDL command was received by the SARM from the SRP.
csdl_prov	int	Number of times the CSDL command was provisioned.
csdl_comp	int	Number of times the CSDL command successfully completed.
csdl_fail	int	Number of times the CSDL command failed.
comp_time_avg	float	Average completion time of the ASDL command.
comp_time_min	float	Minimum completion time of the ASDL command.
comp_time_max	float	Maximum completion time of the ASDL command.
asdl_comp_avg	float	Average number of completed ASDLs for the CSDL command.
asdl_comp_min	int	Minimum number of completed ASDLs for the CSDL command.
asdl_comp_max	int	Maximum number of completed ASDLs for the CSDL command.
asdl_skip_avg	float	Average number of skipped ASDLs for the CSDL command.
asdl_skip_min	int	Minimum number of skipped ASDLs for the CSDL command.

Table 4-173 (Cont.) ADM_csdl_stats, PSP_csdl_stats Results

Name	Datatype	Description
asdl_skip_max	int	Maximum number of skipped ASDLs for the CSDL command.
csdl_parm_avg	float	Average number of CSDL parameters on the CSDL command.
csdl_parm_min	int	Minimum number of CSDL parameters on the CSDL command.
csdl_parm_max	int	Maximum number of CSDL parameters on the CSDL command.

PSP_db_admin

This function purges all admin performance data that have been stored for more than a specified number of days. The default value of a_days is 3 days if it is not provided.

Affected tables:

- tbl_perf_order
- tbl_perf_ne_asdl
- tbl_perf_ne
- tbl_perf_csdl
- tbl_perf_asdl

For more information about using functions, see "Oracle Execution Examples."

Table 4-174 PSP_db_admin Parameters

Name	Description	Req'd	(I)nput/ (O)utput
days	Specifies the age (in days) of performance datato delete. All data older than the specified number of days is deleted.	Yes	I

ADM_ne_asdl_stats, PSP_ne_asdl_stats

These RPCs query NE statistics from tbl_perf_ne_asdl.

For more information about using functions, see "Oracle Execution Examples."

Table 4-175 ADM_ne_asdl_stats, PSP_ne_asdl_stats Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
sarm	SARM where data originated.	No	I/O

Table 4-175 (Cont.) ADM_ne_asdl_stats, PSP_ne_asdl_stats Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	No	I
asdl_cmd	ASDL command.	No	I
user_type	User defined error type.	No	I
from_dts, to_dts	Historical inquiry timestamp range.	No	I

Table 4-176 ADM_ne_asdl_stats, PSP_ne_asdl_stats Results

Name	Datatype	Description
sarm	char(8)	SARM where data originated.
update_dts	datetime	Update timestamp.
host_clli	TYP_mcli	Host NE identifier.
asdl_cmd	TYP_asdl_cmd	ASDL command.
user_type	TYP_code	User defined error type.
value	int	User exit type count.

ADM_ne_stats, PSP_ne_stats

These RPCs query NE statistics from tbl_perf_ne.

For more information about using functions, see "Oracle Execution Examples."

Table 4-177 ADM_ne_stats, PSP_ne_stats Parameters

Name	Description	Req'd	(I)nput/(O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
sarm	SARM where data originated.	No	I/O
host_clli	The host NE identifier of an NE or SRP.	No	I
from_dts, to_dts	Historical inquiry timestamp range.	No	I

Table 4-178 ADM_ne_stats, PSP_ne_stats Results

Name	Datatype	Description
sarm	char(8)	SARM where data originated.
update_dts	datetime	Update timestamp.
nep_svr_cd	TYP_code	The NEP managing the secondary pool of devices.
host_clli	TYP_mcli	Host NE identifier.
tech	TYP_tech	NE technology.
sftwr_load	TYP_load	NE software version.

Table 4-178 (Cont.) ADM_ne_stats, PSP_ne_stats Results

Name	Datatype	Description
state	varchar(25)	Current NE status.
estimate	int	ASDL estimate for the NE.
pending	int	Pending queue size.
in_progress	int	In progress queue size.
connect_count	int	Number of connections to NE.
retry_count	int	Retry queue size.
asdl_qtm_avg	float	Average time for an ASDL in the pending queue.
asdl_qtm_min	int	Minimum time for an ASDL in the pending queue.
asdl_qtm_max	int	Maximum time for an ASDL in the pending queue.
tot_ne_avail	float	Total time in seconds for which the NE is available. NE available time is defined as the time from when NE became available (successful connect), to the time that a disconnect was sent to the NE.
cur_ne_avail	float	Amount of time for which the NE has been available. If the NE is currently down, this time will be 0.
ne_usage	float	Percentage of time that the NE was used for provisioning activities.
num_asdl_comp	int	Number of ASDLs completed at the NE.
num_asdl_fail	int	Number of ASDLs that failed at the NE.
num_asdl_retry	int	Number of ASDLs that were retried at the NE.
tot_maint_tm	float	Total time for which NE has been in maintenance mode.
cur_maint_tm	float	Current time for which the NE is in maintenance mode.
num_asdl_rcvd	int	Total number of ASDLs received by the NE.
num_asdl_xfer	int	Total number of ASDLs transferred.
asdl_comp_tm_avg	float	Average completion time for an ASDL command.
asdl_comp_tm_min	float	Minimum completion time for an ASDL command.
asdl_comp_tm_max	float	Maximum completion time for an ASDL command.
pend_q_avg	float	Average number of ASDLs in the pending queue.
pend_q_max	int	Maximum number of ASDLs in the pending queue.

ADM_order_stats, PSP_order_stats

These RPCs query order statistics from tbl_perf_order.

For more information about using functions, see "Oracle Execution Examples."

Table 4-179 ADM_order_stats, PSP_order_stats Parameters

Name	Description	Req'd	(I)nput/(O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
sarm	SARM where data originated.	No	I/O
org_unit	Organization unit.	No	I
ord_type	Order type.	No	I
from_dts, to_dts	Historical inquiry timestamp range.	No	I

Table 4-180 ADM_order_stats, PSP_order_stats Results

Name	Datatype	Description
sarm	char(8)	SARM where data originated.
update_dts	datetime	Update timestamp.
org_unit	TYP_org_unit	Organization unit specified on order.
org_type	TYP_grp_cd	Order type.
global_p_avg	Float	Average number of global parameters.
global_p_min	int	Minimum number of global parameters.
global_p_max	int	Maximum number of global parameters.
ord_rec	int	Number of orders recorded.
ord_cancelled	int	Number of orders cancelled.
ord_rcvd	int	Number of order operations received. This value can be greater than the number of orders saved in the database.
ord_future	int	Number of future orders received.
ord_imm_fail	int	Number of orders that have failed.
ord_imm_comp	int	Number of orders that have been completed.
ord_tran_err	int	Number of orders that have translation errors.
ord_tmout	int	Number of orders timed-out.
ord_auto_rback	int	Number of orders that have been rolled back automatically.
ord_update	int	Number of orders that were updated after the original order was sent.
ord_collision	int	Number of order collisions.
ord_rtng_err	int	Number of orders that had routing errors.
latency_avg	float	Average order latency.
latency_min	int	Minimum order latency.
latency_max	int	Maximum order latency.
ord_fut_fail	int	The number of future-dated work orders that have failed.
ord_fut_comp	int	The number of future-dated work orders that have completed.

Switch activation and deactivation

A system administrator can activate and deactivate NE access using ASAP.

Once the system has been taken down and then brought back up, the NEs are automatically enabled.

SSP_ne_control

This function controls NE access by ASAP.

For more information about using functions, see "Oracle Execution Examples."

Table 4-181 SSP_ne_control Parameters

Name	Description	Req'd	(I)nput/ (O)utput
host_clli	The host NE identifier of an NE or SRP.	Yes	I
activate	Yes/no to activate or deactivate access to the NE.	Yes	I

C++ SRP API library

This chapter describes the ASAP C++ SRP API functions implemented in the libcsolsrp. This library provides an object-oriented API interface to the SARM to submit work orders and receive information about the progress of work orders, including events generated by the SARM.

SRP_Context class

The **SRP_Context** class initializes and starts the SRP. You must call the getInstance function with arguments before calling any functions in the libcsolsrp. This object has only one instance in the SRP.

Synopsis

```
class SRP_Context
{
public:

static SRP_Context * getInstance(void);
static SRP_Context * getInstance(int argc, char **argv,
SRP_EventInterfaceFactory *eventInterfaceFactory);
static SRP_WoUtils *getWoUtils(void);
static SRP_EventInterfaceFactory *getEventInterfaceFactory(void);
CS_RETCODE getUnId(WORK_ORD_NUM unIdType,
CS_INT &unIdValue, CS_INT nonBlock);
CS_RETCODE extSysAvailable(ASAP_EXTSYS_ID extSysId,
CS_INT nonBlock);
;
```

Public methods

The following are the public methods.

getInstance

Returns the object of this class. Returns 0 if failed. This can only be used after the one with the arguments is called.

Syntax:

```
static SRP_Context * getInstance(void);
```

getInstance

Returns the object of this class. If the object does not exist, a new object of the class is created and returned. The **argc** and **argv** arguments make an analogy to the arguments of **main()**.

For more information, see "SRP_EventInterfaceFactory class."

The eventInterfaceFactory is an object of the SRP_EventInterfaceFactory class. You must create an object of the SRP_EventInterfaceFactory class before calling or using SRP_Context class. It returns 0 if failed.

Syntax:

```
static SRP_Context * getInstance(int argc, char **argv, SRP_EventInterfaceFactory
*eventInterfaceFactory);
```

getWoUtils

Returns the object of the SRP_WoUtils class. It returns 0 if failed.

Syntax:

```
static SRP_WoUtils *getWoUtils(void);
```

getUnId

Returns the unId value of a specified unIdType from the SARM database. If the nonBlock is set to 1, this function returns CS_FAIL when a Sybase RPC communication problem occurs. If the nonBlock is set as 0, this function does not return CS_FAIL even if a Sybase RPC communication problem occurs. This function returns CS_SUCCEED if successful; CS_FAIL if failed.

Syntax:

```
CS_RETCODE getUnId(WORK_ORD_NUM unIdType, CS_INT &unIdValue, CS_INT nonBlock);
```

extSysAvailable

This function notifies SARM that a specified external system has become available again. If the nonBlock is set as 1, this function returns CS_FAIL when a Sybase RPC communication problem occurs. If the nonBlock is set as 0, this function does not return CS_FAIL even if a

Sybase RPC communication problem occurs. This function returns CS_SUCCEED if successful; CS_FAIL if failed.

Syntax:

```
CS_RETCODE extSysAvailable(ASAP_EXTSYS_ID extSysId, CS_INT nonBlock);
```

getEventInterfaceFactory

Returns the object of the **SRP_EventInterfaceFactory** class. The user application must pass the object when it creates the object of the **SRP_Context** class.

Syntax:

```
static SRP_EventInterfaceFactory *getEventInterfaceFactory(void);
```

SRP_Parameter class

You can use the SRP_Parameter class to create and manipulate a parameter object. The object of this class is assigned to a CSDL or work order.

Synopsis

```
class SRP_Parameter
{
public:

    SRP_Parameter(const CSDL_PARAM_LABEL label, const CSDL_PARAM_VALUE value);
    SRP_Parameter(SRP_CSDL *csdl, const CSDL_PARAM_LABEL label,
    const CSDL_PARAM_VALUE value);
    SRP_Parameter(SRP_WO *wo, const CSDL_PARAM_LABEL label,
    const CSDL_PARAM_VALUE value);
    SRP_Parameter(const SRP_Parameter &parameter);    // new!
    ~SRP_Parameter(void);
    inline void getParameterLabel(CSDL_PARAM_LABEL &label) const;
    inline void setParameterLabel(const CSDL_PARAM_LABEL label);
    inline void getParameterValue(CSDL_PARAM_VALUE &value) const;
    inline void setParameterValue(const CSDL_PARAM_VALUE value);
    CS_BOOL operator==(const SRP_Parameter &parameter) const;
    SRP_Parameter& operator=(const SRP_Parameter &parameter);
    void print(std::ofstream &outFile);
    inline CS_RETCODE lock(void);
    inline CS_RETCODE unlock(void);

private:
    SRP_Parameter(void) {};
    CSDL_PARAM_LABEL m_label;
    CSDL_PARAM_VALUE m_value;
    ASC_Mutex *m_mutex;
    Diagnosis m_diag;    Event m_event;
};
```

Constructors

Use this constructor to provide the label and value arguments for the parameter object.

Syntax:

```
SRP_Parameter(const CSDL_PARAM_LABEL label, const CSDL_PARAM_VALUE value);
```

In addition to providing the label and value arguments, this constructor function inserts the parameter object into the specified CSDL object.

Syntax:

```
SRP_Parameter(SRP_CSDL *csdl, const CSDL_PARAM_LABEL label, const CSDL_PARAM_VALUE
value);
```

Provides the label and value arguments. In addition, this constructor function inserts the parameter object into the specified work order object as a global parameter.

Syntax:

```
SRP_Parameter(SRP_WO *wo, const CSDL_PARAM_LABEL label, const CSDL_PARAM_VALUE value);
```

Public methods

The following are the public methods.

getParameterLabel

Returns the parameter label.

Syntax:

```
void getParameterLabel(CSDL_PARAM_LABEL &label) const;
```

setParameterLabel

Sets the parameter label. You can use this function to modify the object.

Syntax:

```
void setParameterLabel(const CSDL_PARAM_LABEL label);
```

getParameterValue

Returns the parameter value.

Syntax:

```
void getParameterValue(CSDL_PARAM_VALUE &value) const;
```

setParameterValue

Sets the parameter value. You can use this function to modify the object.

Syntax:

```
void setParameterValue(const CSDL_PARAM_VALUE value);
```

operator==

This operator compares two parameter objects. If the parameter labels are the same, this function returns CS_TRUE. If the labels are different, this function returns CS_FALSE.

Syntax:

```
CS_BOOL operator==(const SRP_Parameter &parameter) const;
```

print

This function writes the label and value of the current parameter object into the output file that you specify. You must open the output file before calling this function.

Syntax:

```
void print(std::ofstream &outFile);
```

lock

This function locks the current parameter object. Returns CS_SUCCEED if successful.

Syntax:

```
CS_RETCODE lock(void);
```

unlock

This function unlocks the current parameter object. Returns CS_SUCCEED if successful.

Syntax:

```
CS_RETCODE unlock(void);
```

SRP_CSDL class

The SRP_CSDL class is used to create and manipulate a CSDL object. The object of this class is assigned to a work order object.

Synopsis

```
class SRP_CSDL
{
public:

    SRP_CSDL(
    const CSDL_CMD_ST csdlCmd,
    const ASAP_CSDL_TRAN_STATUS csdlStatus,
    ASAP_CSDL_DESC csdlDesc = "",
    ASAP_CSDL_ID csdlId = 0);
    SRP_CSDL(
    SRP_WO *wo,
    const CSDL_CMD_ST csdlCmd,
    const ASAP_CSDL_TRAN_STATUS csdlStatus,
    ASAP_CSDL_DESC csdlDesc = "",
    ASAP_CSDL_ID csdlId = 0);
    SRP_CSDL(const SRP_CSDL &csdl);
    ~SRP_CSDL(void);
    inline ASAP_CSDL_ID getCsdlId(void) const;
    inline void setCsdlId(const ASAP_CSDL_ID csdlId);
    inline void getCsdlCmd(CSDL_CMD_ST &csdlCmd) const;
    inline void setCsdlCmd(const CSDL_CMD_ST csdlCmd);
    inline ASAP_CSDL_TRAN_STATUS getCsdlStatus(void) const;
    inline void setCsdlStatus(const ASAP_CSDL_TRAN_STATUS csdlStatus);
    inline void getCsdlDesc(ASAP_CSDL_DESC &csdlDesc) const;
```

```
    inline void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);
    SRP_CSDL& operator=(const SRP_CSDL& csdl);
    CS_RETCODE addParameter(SRP_Parameter *parameter);
    CS_INT getParameterCount(void);
    void print(std::ofstream &outFile);
    void deleteAllParameters(void);
    SRP_Parameter * findParameter(const CSDL_PARAM_LABEL label);
    SRP_Parameter * findParameter(SRP_Parameter *parameter);
    SRP_Parameter * findParameter(CS_INT sequence);
    SRP_Parameter *  removeParameter(const CSDL_PARAM_LABEL label);
    SRP_Parameter *  removeParameter(SRP_Parameter *parameter);
    inline std::set<SRP_Parameter*, SRPParamComparator>::iterator
    find_by_label(std::set<SRP_Parameter*, SRPParamComparator> &container, const
CSDL_PARAM_LABEL label);
    inline CS_RETCODE lock(void);
    inline CS_RETCODE unlock(void);
    inline std::set<SRP_Parameter*, SRPParamComparator>* getParameters(void) const;

private:
    friend class SRP_RPCSubmitInterface;
    SRP_CSDL(void) {};
    ASC_Mutex *m_mutexParameters;
    CSDL_CMD_ST m_csdlCmd;
    ASAP_CSDL_TRAN_STATUS m_csdlStatus;
    ASAP_CSDL_DESC m_csdlDesc;
    ASAP_CSDL_ID m_csdlId;
    std::set<SRP_Parameter*, SRPParamComparator> m_parameters;
    ASC_Mutex *m_mutex;
    Diagnosis m_diag;
    Event m_event;
};
```

Constructors

This constructor function must include the CSDL command and status. The CSDL description and CSDL ID are optional.

Syntax:

```
SRP_CSDL(const CSDL_CMD_ST csdlCmd, const ASAP_CSDL_TRAN_STATUS csdlStatus,
ASAP_CSDL_DESC csdlDesc = "", ASAP_CSDL_ID csdlId = 0);
```

This constructor function must include the CSDL command and status. The CSDL description and CSDL ID are optional. In addition, this function inserts the CSDL object into the specified work order object.

Syntax:

```
SRP_CSDL(SRP_WO *wo, const CSDL_CMD_ST csdlCmd, const ASAP_CSDL_TRAN_STATUS
csdlStatus, ASAP_CSDL_DESC csdlDesc = "", ASAP_CSDL_ID csdlId = 0);
```

This constructor function must include an object of type SRP_CSDL.

Syntax:

```
SRP_CSDL(const SRP_CSDL &csdl);
```

Public methods

The following are the public methods.

getCsdlId

Returns the CSDL unique identifier.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

setCsdlId

Sets the CSDL ID is used to modify the object.

Syntax:

```
void setCsdlId(const ASAP_CSDL_ID csdlId);
```

getCsdlCmd

Returns the CSDL command.

Syntax:

```
void getCsdlCmd(CSDL_CMD_ST &csdlCmd) const;
```

setCsdlCmd

Sets the CSDL value and is used to modify the object.

Syntax:

```
void setCsdlCmd(const CSDL_CMD_ST csdlCmd);
```

getCsdlStatus

Returns the CSDL status. Possible values are:

- ASAP_CSDL_INITIAL – The CSDL command is ready to be provisioned.
- ASAP_CSDL_HELD – The CSDL command is held from provisioning.
- ASAP_CSDL_MANUAL_TRAN – CSDL command to be manually translated. This causes the work order not to be provisioned.
- ASAP_CSDL_TRAN_ERR – CSDL command translation error.

Syntax:

```
ASAP_CSDL_TRAN_STATUS getCsdlStatus(void) const;
```

setCsdlStatus

Sets the CSDL status.

Syntax:

```
void setCsdlStatus(const ASAP_CSDL_TRAN_STATUS csdlStatus);
```

getCsdlDesc

Returns the CSDL description.

Syntax:

```
void getCsdlDesc(ASAP_CSDL_DESC &csdlDesc) const;
```

setCsdlDesc

Sets the CSDL description.

Syntax:

```
void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);
```

addParameter

Adds a parameter to the CSDL.

Syntax:

```
CS_RETCODE addParameter(SRP_Parameter *parameter);
```

getParameterCount

Returns the number of parameters in the CSDL.

Syntax:

```
CS_INT getParameterCount(void);
```

print

Writes the contents of the CSDL into the output file.

Syntax:

```
void print(std::ofstream &outFile);
```

deleteAllParameters

Removes all parameters from the CSDL and deletes the parameters. After the execution of this command, the result of referencing the deleted parameters is unpredictable.

Syntax:

```
void deleteAllParameters(void);
```

findParameter

Finds a specified parameter from the CSDL by the parameter label. The operation does not remove the parameter from the C++ SRP API.

Syntax:

```
SRP_Parameter * findParameter(const CSDL_PARAM_LABEL label);
```

Finds a specified parameter from the CSDL by the parameter object.

Syntax:

```
SRP_Parameter * findParameter(SRP_Parameter *parameter);
```

Finds a specified parameter from the CSDL by the sequence of the parameters in the CSDL.

Syntax:

```
SRP_Parameter * findParameter(CS_INT sequence);
```

removeParameter

Removes a specified parameter from the CSDL by the parameter label. This operation does not delete the parameter removed.

Syntax:

```
SRP_Parameter * removeParameter(const CSDL_PARAM_LABEL label);
```

Removes a specified parameter from the CSDL by parameter object. This operation does not delete the parameter removed.

Syntax:

```
SRP_Parameter * removeParameter(SRP_Parameter *parameter);
```

lock

Locks the current CSDL. The operation is blocked until it obtains the lock.

Syntax:

```
CS_RETCODE lock(void);
```

find_by_label

Finds a specified parameter from the CSDL by the parameter label. This function takes the container, **set**, as an argument and returns the iterator to the specified parameter.

Syntax:

```
find_by_label(std::set<SRP_Parameter*, SRPParamComparator> &container, const
CSDL_PARAM_LABEL label);
```

unlock

Unlocks the current CSDL. You must always use the **lock()** method before using **unlock()** in a thread.

Syntax:

```
CS_RETCODE unlock(void);
```

getParameters

Returns a set of parameters corresponding to the CSDL.

```
inline std::set<SRP_Parameter*, SRPParamComparator>* getParameters(void) const;
```

SRP_WO class

The **SRP_WO** class is used to create and manipulate a work order object. The **SRP_WO** class provides all services for the upstream application to submit a service request to ASAP. The upstream application must generate a work order object in order to complete a service request. This class provides work order generation and submission functions.

Synopsis

```
class SRP_WO
{
public:

SRP_WO(void);
SRP_WO(const WORK_ORD_NUM woId);
SRP_WO(
const WORK_ORD_NUM woId,
const CS_DATETIME dueDate,
CS_INT operation = ASAP_CMD_WO_UPDATE,
CS_INT asdlTimeout = 0,
SRP_USERID userId = "",
SRP_PASSWORD password = "",
ASAP_SRQ_PRIORITY priority = ASAP_SRQ_NORMAL_PRIO,
DAT_ACTN_ST srqAction = ADD_ACTN,
ASAP_ORG_UNIT orgUnit = "",
SRP_LOGIN origin = "",
WORK_ORD_NUM parentWo = "",
CS_INT woTimeout = USE_DEFAULT,
CS_INT retry = USE_DEFAULT,
CS_INT retryInt = USE_DEFAULT,
ASAP_BOOL_PROP_TYPE rback = "D",
ASAP_BOOL_PROP_TYPE delayFail = "N",
CS_INT delayFailThreshold = 0
);
// Work Order General Transaction Member Functions


// Property processing functions
void getWoId(WORK_ORD_NUM &workId) const;
void setWoId(const WORK_ORD_NUM workId);

void getDueDate(CS_DATETIME &dueDate) const;
CS_RETCODE getDueDate(CS_CHAR *dueDate) const;
void setDueDate(const CS_DATETIME dueDate);
void setDueDate(const CS_INT dtdays, const CS_INT dttime);
CS_RETCODE setDueDate(CS_CHAR *dueDate) const;

CS_INT getOperation(void) const;
void setOperation(const CS_INT operation);

void getMisc(ASAP_WO_MISC &misc) const;
void setMisc(const ASAP_WO_MISC misc);

void getOrgUnit(ASAP_ORG_UNIT &orgUnit) const;
void setOrgUnit(const ASAP_ORG_UNIT orgUnit);

void getOrigin(SRP_LOGIN &origin) const;
```

```
void setOrigin(const SRP_LOGIN origin);

CS_INT getEstimate(void) const;
void setEstimate (const CS_INT estimate);

CS_INT getStatus(void) const;
void setStatus (const CS_INT status);

CS_INT getAsdlTimeout(void) const;
void setAsdlTimeout(const CS_INT asdlTimeout);

void getUserId(SRP_USERID &userId) const;
void setUserId(const SRP_USERID userId);

void getPassword(SRP_PASSWORD &password) const;
void setPassword(const SRP_PASSWORD password);

void getPriority(ASAP_SRQ_PRIORITY &priority) const;
void setPriority(const ASAP_SRQ_PRIORITY priority);

void getSrqAction(DAT_ACTN_ST &srqAction) const;
void setSrqAction(const DAT_ACTN_ST srqAction);

void getParentWo(WORK_ORD_NUM &parentWo) const;
void setParentWo(const WORK_ORD_NUM parentWo);

CS_INT getWoTimeout(void) const;
void setWoTimeout(const CS_INT woTimeout);

CS_INT getRetry(void) const;
void setRetry(const CS_INT retry);

CS_INT getRetryInt(void) const;
void setRetryInt(const CS_INT retryInt);

void getRback(ASAP_BOOL_PROP_TYPE &rback) const;
void setRback(const ASAP_BOOL_PROP_TYPE rback);

void getDelayFail(ASAP_BOOL_PROP_TYPE &delayFail) const;
void setDelayFail(const ASAP_BOOL_PROP_TYPE delayFail);

CS_INT getDelayFailThreshold(void) const;
void setDelayFailThreshold(const CS_INT delayFailThreshold);

void getBatchGroup(WORK_ORD_NUM &batchGroup) const;
void setBatchGroup(const WORK_ORD_NUM batchGroup);

void getExtSysId(ASAP_EXTSYS_ID &extSysId) const;
void setExtSysId(const ASAP_EXTSYS_ID extSysId);

VOIDPTR getUserData(void) const;
void setUserData(const VOIDPTR userData);

void getApplName(APPL_NAME & applName) const;
void setApplName(const APPL_NAME applName);

CS_RETCODE getProperty(CS_INT property, CS_CHAR * value);
CS_RETCODE getProperty(CS_INT property, CS_INT &value);
CS_RETCODE getProperty(CS_INT property, CS_DATETIME &value);

CS_RETCODE setProperty(CS_INT property, const CS_CHAR * value);
```

```
CS_RETCODE setProperty(CS_INT property, const CS_INT value);
CS_RETCODE setProperty(CS_INT property, const CS_DATETIME value);

// Transaction processing functions
CS_RETCODE addCsdl(SRP_CSDL *csdl);
CS_RETCODE addGlobalParameter(SRP_Parameter *parameter);

CS_RETCODE restore(void);

CS_INT getGlobalParameterCount(void);
CS_INT getCsdlCount(void);

void print(ofstream &outFile);
//Work Order Submission member functions
CS_RETCODE submit(CS_INT noneBlock = 0);
CS_RETCODE deleteInSarm(CS_INT noneBlock = 0);
CS_RETCODE changeStatus(CS_INT noneBlock = 0);

// Special delete functions
void deleteAll(void);
SRP_Parameter * findGlobalParameter(const CSDL_PARAM_LABEL label);
SRP_Parameter * findGlobalParameter(SRP_Parameter *parameter);
SRP_Parameter * findGlobalParameter(CS_INT sequence);

SRP_Parameter * removeGlobalParameter(const
CSDL_PARAM_LABEL label);
SRP_Parameter * removeGlobalParameter(SRP_Parameter
*parameter);

SRP_CSDL * findCsdl(const CSDL_CMD_ST csdlCmd);
SRP_CSDL * findCsdl(const ASAP_CSDL_ID csdlId);
SRP_CSDL * findCsdl(SRP_CSDL *csdl);
SRP_CSDL * findCsdlBySequence(CS_INT sequence);

SRP_CSDL * removeCsdl(const CSDL_CMD_ST csdlCmd);
SRP_CSDL * removeCsdl(const ASAP_CSDL_ID csdlId);
SRP_CSDL * removeCsdl(SRP_CSDL *csdl);

CS_RETCODE lock(void);
CS_RETCODE unlock(void);

};
```

Constructors

This constructor invokes a work order object.

Syntax:

```
SRP_WO(void);
```

This constructor invokes a work order object. This function requires specified work order identification.

Syntax:

```
SRP_WO(const WORK_ORD_NUM woId);
```

Work order properties

```
SRP_WO(const WORK_ORD_NUM woId,

const CS_DATETIME dueDate,
CS_INT operation = ASAP_CMD_WO_UPDATE,
CS_INT asdlTimeout = 0,
SRP_USERID userId = "",
SRP_PASSWORD password = "",
ASAP_SRQ_PRIORITY priority =
ASAP_SRQ_NORMAL_PRIO,
DAT_ACTN_ST srqAction = ADD_ACTN,
ASAP_ORG_UNIT orgUnit = "",
SRP_LOGIN origin = "",
WORK_ORD_NUM parentWo = "",
CS_INT woTimeout = USE_DEFAULT,
CS_INT retry = USE_DEFAULT,
CS_INT retryInt = USE_DEFAULT,
ASAP_BOOL_PROP_TYPE rback = "D",
ASAP_BOOL_PROP_TYPE delayFail = "N",
CS_INT delayFailThreshold = 0

);
```

This constructor requires the following attributes which assign all work order properties:

- **woId:** Work order identification.
- **dueDate:** The due date and time for work order provisioning.
- **operation:** The operation that the SARM is to perform with the work order. The values are defined in the asap_core.h.
- **asdlTimeout:** ASDL timeout.
- **userId:** The user ID to be used for security authorization.
- **password:** The password to be used for security authorization.
- **priority:** The provisioning priority of the Service Request. The values are defined in the asap_core.h.
- **srqAction:** The action type of the Service Request being sent. The values are defined in the libcsolsrp.h.
- **orgUnit:** Organization Unit associated with the work order. This is the ID of a person or group to whom notification should be transmitted should an error or exception occur on the work order.
- **origin:** Identifies the Host system user who created the work order.
- **parentWo:** Parent work order for work order dependencies.
- **woTimeout:** Work order timeout. If a work order is in progress more than the set number of seconds, then the SARM will fail the work order.
- **retry:** The number of times to retry the work order.
- **retryInt:** The retry interval on the work order.
- **rback:** Rollback flag on the work order.
- **delayFail:** Treat Hard ASDL errors in the SARM as Delayed Errors.

- **delayFailThreshold:** If delayFail is set, this field represents the number of such delayed errors that must occur for the SARM to stop the work order processing.

Most of these attributes have a default value that is used if you do not provide one. To modify the properties after a work order object is created, you can also use the **set** operations below.

Public methods

The following are the public methods.

getWoId

Returns the work order ID.

Syntax:

```
void getWoId(WORK_ORD_NUM &workId) const;
```

setWoId

Sets a specified work order ID.

Syntax:

```
void setWoId(const WORK_ORD_NUM workId);
```

getDueDate

Returns the duration date by the CS_DATETIME data type.

Syntax:

```
void getDueDate(CS_DATETIME &dueDate) const;
```

Returns the due date by the CS_CHAR data type.

Syntax:

```
CS_RETCODE getDueDate(CS_CHAR *dueDate) const;
```

setDueDate

Sets the CSDL description by the CS_DATETIME.

Syntax:

```
void setDueDate(const CS_DATETIME dueDate);
```

Sets the due date and time by the dtdays and dttime.

Syntax:

```
void setDueDate(const CS_INT dtdays, const CS_INT dttime);
```

Sets the due date and time by CS_CHAR data type.

Syntax:

```
CS_RETCODE setDueDate(CS_CHAR *dueDate) const;
```

getOperation

Returns the operation.

Syntax:

```
CS_INT getOperation(void) const;
```

setOperation

Sets a specified operation.

Syntax:

```
void setOperation(const CS_INT operation);
```

getMisc

Returns the miscellaneous information.

Syntax:

```
void getMisc(ASAP_WO_MISC &misc) const;
```

setMisc

Sets miscellaneous information.

Syntax:

```
void setMisc(const ASAP_WO_MISC misc);
```

getOrgUnit

Returns the organization unit.

Syntax:

```
void getOrgUnit(ASAP_ORG_UNIT &orgUnit) const;
```

setOrgUnit

Sets a specified organization unit.

Syntax:

```
void setOrgUnit(const ASAP_ORG_UNIT orgUnit);
```

getOrigin

Returns the origin.

Syntax:

```
void getOrigin(SRP_LOGIN &origin) const;
```

setOrigin

Sets a specified origin.

Syntax:

```
void setOrigin(const SRP_LOGIN origin);
```

getEstimate

Returns the estimated amount of time (in seconds) for a work order to be completed by the SARM. The value is the total time from the work order being initially received by the SARM. The upstream application can set the estimate using the setEstimate function.

Syntax:

```
CS_INT getEstimate(void) const;
```

getStatus

Returns the status of a specified work order transaction, returned by the SARM. These values are defined in **asap_core.h**. This value must be checked after the submission to determine if the submission is successful.

Syntax:

```
CS_INT getStatus(void) const;
```

getAsdlTimeout

Returns the ASDL timeout.

Syntax:

```
CS_INT getAsdlTimeout(void) const;
```

setAsdlTimeout

Sets a specified ASDL timeout.

Syntax:

```
void setAsdlTimeout(const CS_INT asdlTimeout);
```

getUserId

Returns the user ID.

Syntax:

```
void getUserId(SRP_USERID &userId) const;
```

setUserId

Sets the user ID.

Syntax:

```
void setUserId(const SRP_USERID userId);
```

getPassword

Returns the password.

Syntax:

void getPassword(SRP_PASSWORD &password) const;

setPassword

Sets the password.

Syntax:

```
void setPassword(const SRP_PASSWORD password);
```

getPriority

Returns the priority of the Service Request.

Syntax:

```
void getPriority(ASAP_SRQ_PRIORITY &priority) const;
```

setPriority

Sets the priority of the Service Request.

Syntax:

```
void setPriority(const ASAP_SRQ_PRIORITY priority);
```

getSrqAction

Returns the action type of the Service Request.

Syntax:

```
void getSrqAction(DAT_ACTN_ST &srqAction) const;
```

setSrqAction

Sets the specified action type of the Service Request.

Syntax:

```
void setSrqAction(const DAT_ACTN_ST srqAction);
```

getParentWo

Returns the parent work order for work order dependencies.

Syntax:

```
void getParentWo(WORK_ORD_NUM &parentWo) const;
```

setParentWo

Sets the parent work order.

Syntax:

```
void setParentWo(const WORK_ORD_NUM parentWo);
```

getWoTimeout

Returns the work order timeout.

Syntax:

```
CS_INT getWoTimeout(void) const;
```

setWoTimeout

Sets the work order timeout.

Syntax:

```
void setWoTimeout(const CS_INT woTimeout);
```

getRetry

Returns the number of times to retry the work order.

Syntax:

```
CS_INT getRetry(void) const;m
```

setRetry

Sets the number of retries.

Syntax:

```
void setRetry(const CS_INT retry);
```

getRetryInt

Returns the retry interval.

Syntax:

```
CS_INT getRetryInt(void) const;
```

setRetryInt

Sets a specified retry interval.

Syntax:

```
void setRetryInt(const CS_INT retryInt);
```

getRback

Returns the rollback flag.

Syntax:

```
void getRback(ASAP_BOOL_PROP_TYPE &rback) const;
```

setRback

Sets the rollback flag.

Syntax:

```
void setRback(const ASAP_BOOL_PROP_TYPE rback);
```

getDelayFail

Returns the delay fail flag.

Syntax:

```
void getDelayFail(ASAP_BOOL_PROP_TYPE &delayFail) const;
```

setDelayFail

Sets the delay fail flag.

Syntax:

```
void setDelayFail(const ASAP_BOOL_PROP_TYPE delayFail);
```

getDelayFailThreshold

Returns the delay fail threshold.

Syntax:

```
CS_INT getDelayFailThreshold(void) const;
```

setDelayFailThreshold

Sets the delay fail threshold.

Syntax:

```
void setDelayFailThreshold(const CS_INT delayFailThreshold);
```

getBatchGroup

Returns the batch group.

Syntax:

```
void getBatchGroup(WORK_ORD_NUM &batchGroup) const;
```

setBatchGroup

Sets the batch group.

Syntax:

```
void setBatchGroup(const WORK_ORD_NUM batchGroup);
```

getExtSysId

Returns the external system ID for this work order.

Syntax:

```
void getExtSysId(ASAP_EXTSYS_ID &extSysId) const;
```

setExtSysId

Sets a specified external system ID.

Syntax:

```
void setExtSysId(const ASAP_EXTSYS_ID extSysId);
```

getUserData

Returns the user-defined data segment in the work order.

Syntax:

```
VOIDPTR getUserData(void) const;
```

setUserData

Sets a user-defined data segment.

Syntax:

```
void setUserData(const VOIDPTR userData);
```

getApplName

Returns the application name.

Syntax:

```
void getApplName(APPL_NAME & applName) const;
```

setApplName

Sets a specified application name in the work order. If you do not define a specified application name, the default is the current SRP name.

Syntax:

```
void setApplName(const APPL_NAME applName);
```

getProperty

Returns the value of a specified property. This is one of the overloaded operations. The properties are defined in **libcsolsrp.h**.

Syntax:

```
CS_RETCODE getProperty(CS_INT property, CS_CHAR * value);
```

Returns the value of a specified property. This is one of the overloaded operations. The properties are defined in **libcsolsrp.h**.

Syntax:

```
CS_RETCODE getProperty(CS_INT property, CS_INT &value);
```

Returns the value of a specified property. This is one of the overloaded operations. The properties are defined in **libcsolsrp.h**.

Syntax:

```
CS_RETCODE getProperty(CS_INT property, CS_DATETIME &value);
```

setProperty

Sets the value for a specified property whose data type is CS_CHAR.

Syntax:

```
CS_RETCODE setProperty(CS_INT property, const CS_CHAR * value);
```

Sets the value for a specified property whose data type is CS_INT.

Syntax:

```
CS_RETCODE setProperty(CS_INT property, const CS_INT value);
```

Sets the value for a specified property whose data type is CS_DATETIME.

Syntax:

```
CS_RETCODE setProperty(CS_INT property, const CS_DATETIME value);
```

addCsdl

Adds a specified CSDL into the work order.

Syntax:

```
CS_RETCODE addCsdl(SRP_CSDL *csdl);
```

addGlobalParameter

Adds a specified global parameter into the work order.

Syntax:

```
CS_RETCODE addGlobalParameter(SRP_Parameter *parameter);
```

restore

Rebuilds the current work order from the SARM.

Syntax:

```
CS_RETCODE restore(void);
```

getGlobalParameterCount

Returns the number of global parameters.

Syntax:

```
CS_INT getGlobalParameterCount(void);
```

getCsdlCount

Returns the number of CSDLs.

Syntax:

```
CS_INT getCsdlCount(void);
```

print

Prints the content of the work order. The upstream application must provide a file descriptor to print. The file must be opened.

Syntax:

```
void print(ofstream &outFile);
```

submit

Submits the current work order. If nonBlock is set, this function returns a fail when the connection to the SARM is unavailable. If nonBlock is not set, this function retries sending the RPC until it succeeds. The default is block.

Syntax:

```
CS_RETCODE submit(CS_INT nonBlock = 0);
```

deleteInSarm

Deletes the current work order from the SARM. If nonBlock is set, this function returns a fail when the connection to the SARM is unavailable. If nonBlock is not set, this function retries sending the RPC until it succeeds. The default is block.

Syntax:

```
CS_RETCODE deleteInSarm(CS_INT nonBlock = 0);
```

changeStatus

Changes the status of the current work order from the SARM. If nonBlock is set, this function returns a fail when the connection to the SARM is unavailable. If nonBlock is not set, this function retries sending the RPC until it succeeds. The default is block.

Syntax:

```
CS_RETCODE changeStatus(CS_INT nonBlock = 0);
```

deleteAll

Removes all global parameter and CSLD objects from the current work order, and deletes them. The result is unpredictable if you reference those objects after deleting them.

Syntax:

```
void deleteAll(void);
```

findGlobalParameter

Finds a global parameter by parameter label. This operation does not remove the object.

Syntax:

```
SRP_Parameter * findGlobalParameter(const CSDL_PARAM_LABEL label);
```

Finds a global parameter by parameter object. This operation does not remove the object.

Syntax:

```
SRP_Parameter * findGlobalParameter(SRP_Parameter *parameter);
```

Finds a global parameter by sequence of parameter in the work order. This operation does not remove the object.

Syntax:

```
SRP_Parameter * findGlobalParameter(CS_INT sequence);
```

removeGlobalParameter

Removes a global parameter from the work order by parameter label. This operation does not delete the object.

Syntax:

```
SRP_Parameter * removeGlobalParameter(const CSDL_PARAM_LABEL label);
```

Removes a global parameter from the work order by a specified parameter. This operation does not delete the object.

Syntax:

```
SRP_Parameter * removeGlobalParameter(SRP_Parameter *parameter);
```

findCsdl

Finds a CSDL by a specified CSDL command. This operation does not remove the object.

Syntax:

```
SRP_CSDL * findCsdl(const CSDL_CMD_ST csdlCmd);
```

Finds a CSDL by a specified CSDL ID. This operation does not remove the object.

Syntax:

```
SRP_CSDL * findCsdl(const ASAP_CSDL_ID csdlId);
```

Finds a CSDL by a specified CSDL object. This operation does not remove the object.

Syntax:

```
SRP_CSDL * findCsdl(SRP_CSDL *csdl);
```

findCsdlBySequence

Finds a CSDL by the sequence of CSDLs in the work order. This operation does not remove the object.

Syntax:

```
SRP_CSDL * findCsdlBySequence(CS_INT sequence);
```

removeCsdl

Removes a specified CSDL by CSDL command from the work order. This operation does not delete the object.

Syntax:

```
SRP_CSDL * removeCsdl(const CSDL_CMD_ST csdlCmd);
```

Removes a specified CSDL by CSDL ID from the work order. This operation does not delete the object.

Syntax:

```
SRP_CSDL * removeCsdl(const ASAP_CSDL_ID csdlId);
```

Removes a specified CSDL by CSDL object from the work order. This operation does not delete the object.

Syntax:

```
SRP_CSDL * removeCsdl(SRP_CSDL *csdl);
```

lock

Locks the work order object. The operation is blocked until it obtains the lock.

Syntax:

```
CS_RETCODE lock(void);
```

unlock

Releases the work order object. You must always use the **lock()** method before using **unlock()** in a thread.

Syntax:

```
CS_RETCODE unlock(void);
```

SRP_WoUtils class

The **SRP_WoUtils** class is used to lock work orders by work order ID.

Synopsis

```
class SRP_WoUtils
{
public:

CS_RETCODE lockWo(const WORK_ORD_NUM woId);
CS_RETCODE unlockWo(const WORK_ORD_NUM woId);
CS_RETCODE accessWo(const WORK_ORD_NUM woId);

};
```

Public methods

The following are the public methods.

lockWo

This function locks a specified work order by work order ID. Used to synchronize work order submission and work order event handling.

If the work order has been locked already, this operation is blocked until the work order is unlocked. Returns CS_SUCCEED if successfully completed.

Syntax:

```
CS_RETCODE lockWo(const WORK_ORD_NUM woId);
```

unlockWo

Unlocks a specified work order by work order ID. Returns CS_SUCCEED if successfully completed.

Syntax:

```
CS_RETCODE unlockWo(const WORK_ORD_NUM woId);
```

accessWo

This function is called by the work order event notification handlers upon receiving an event for a work order, and before performing any processing of the event involving work order access or update in the SRP database. This helps avoid data integrity problems when the SARM returns an event for the work order before the order is

saved in the SRP database. This function returns CS_SUCCEED if successfully completed; CS_FAIL if failed.

Syntax:

```
CS_RETCODE accessWo(const WORK_ORD_NUM woId);
```

SRP_EventInterfaceFactory class

The **SRP_EventInterfaceFactory** class is used to instantiate the **SRP_EventInterface** objects that you define. This is an abstract class that you must define as a subclass, typically by overwriting **create()**. The C++ SRP API calls this function to create the event interface object that you define.

Synopsis

```
class SRP_EventInterfaceFactory
{
public:

SRP_EventInterfaceFactory(void);

virtual SRP_EventInterface* create(void) = 0;
void setCondition(void *condition);
void* getCondition(void);

protected:

void * m_condition;

Diagnosis m_diag;
Event m_event;

};
```

Constructors

This constructor invokes a **SRP_EventInterfaceFactory** object.

Syntax:

```
SRP_EventInterfaceFactory(void);
```

Public methods

The following are the public methods.

create

This function that you have defined, creates and returns an object of the event interface that you have defined.

Syntax:

```
virtual SRP_EventInterface* create(void) = 0;
```

setCondition

Sets the condition for instantiating an event interface object. You can use this condition to create a different event interface object. For example, you can change the condition from time to time, and when the **create()** operation is invoked, it can check the condition to determine which event interface object must be created.

Syntax:

```
void setCondition(void *condition);
```

getCondition

Returns the condition.

Syntax:

```
void* getCondition(void);
```

SRP_EventInterface class

The **SRP_EventInterface** class is the base class for specific event interface objects. The **SRP_EventInterface** object calls the required event handler that you have defined when the corresponding events are received by the SRP from the SARM.

This is an abstract class. You must create a subclass from the **SRP_EventInterface**, and redefine the work order complete and work order failure event handlers. C++ SRP API provides other types of undefined event handlers. If you define these optional handlers, the event interface object calls the event handler that you have defined with the event object as the argument.

Synopsis

```
class SRP_EventInterface
{
public:
SRP_EventInterface(void);

virtual CS_RETCODE woCompleteHandler (SRP_Event *event) = 0;
virtual CS_RETCODE woFailureHandler (SRP_Event *event) = 0;
virtual CS_RETCODE softErrorHandler (SRP_Event *event);
virtual CS_RETCODE woEstimateHandler(SRP_Event *event);
virtual CS_RETCODE woStartupHandler (SRP_Event *event);
virtual CS_RETCODE woRollbackHandler (SRP_Event *event);
virtual CS_RETCODE neUnknownHandler (SRP_Event *event);
virtual CS_RETCODE woBlockHandler(SRP_Event *event);
virtual CS_RETCODE woTimeOutHandler (SRP_Event *event);
virtual CS_RETCODE neAvailHandler (SRP_Event *event);
virtual CS_RETCODE neUnavailHandler (SRP_Event *event);
virtual CS_RETCODE woAcceptHandler (SRP_Event *event);


protected:

Diagnosis m_diag;
Event m_event;
```

```
};
```

Constructor

Constructor of the class.

Syntax:

```
SRP_EventInterface(void);
```

Public methods

The following are the public methods.

woCompleteHandler

The event handler is for the WO_COMPLETE event. You must redefine this function in the subclass of this class.

Syntax:

```
virtual CS_RETCODE woCompleteHandler (SRP_Event *event);
```

woFailureHandler

The event handler is for the WO_FAILURE event. You must redefine this function in the subclass of this class.

Syntax:

```
virtual CS_RETCODE woFailureHandler (SRP_Event *event);
```

softErrorHandler

The optional event handler is for the SOFT_ERROR event.

Syntax:

```
virtual CS_RETCODE softErrorHandler (SRP_Event *event);
```

woEstimateHandler

The optional event handler is for the WO_ESTIMATE event.

Syntax:

```
virtual CS_RETCODE woEstimateHandler(SRP_Event *event);
```

woStartupHandler

This optional event handler is for the WO_STARTUP event.

Syntax:

```
virtual CS_RETCODE woStartupHandler (SRP_Event *event);
```

woRollbackHandler

The optional event handler is for the WO_ROLLBACK event.

Syntax:

```
virtual CS_RETCODE woRollbackHandler (SRP_Event *event);
```

neUnknownHandler

This optional event handler is for the NE_UNKNOWN event.

Syntax:

```
virtual CS_RETCODE neUnknownHandler (SRP_Event *event);
```

woBlockHandler

The optional event handler is for the WO_BLOCKED event.

Syntax:

```
virtual CS_RETCODE woBlockHandler(SRP_Event *event);
```

woTimeOutHandler

The optional event handler is for the WO_TIMEOUT event.

Syntax:

```
virtual CS_RETCODE woTimeOutHandler (SRP_Event *event);
```

neAvailHandler

The optional event handler is for the NE_AVAIL event.

Syntax:

```
virtual CS_RETCODE neAvailHandler (SRP_Event *event);
```

neUnavailHandler

The optional event handler is for the NE_UNAVAIL event.

Syntax:

```
virtual CS_RETCODE neUnavailHandler (SRP_Event *event);
```

woAcceptHandler

The optional event handler is for the WO_ACCEPT event.

Syntax:

```
virtual CS_RETCODE woAcceptHandler (SRP_Event *event);
```

SRP_Event class

The **SRP_Event** class is a base class for different event classes. This class delivers specified event information to the event handlers. Normally, the upstream application retrieves the content in the event object to check work order provisioning status. It does not need to use the set functions in this class, except to produce a copy of the event object in the event handlers.

> **Note:**
>
> You must not delete this object. C++ SRP API creates and deletes this object.

Synopsis

```
class SRP_Event
{
public:

SRP_Event(void);
void getWoId(WORK_ORD_NUM& woId) const;
CS_INT getEventUnId(void) const;
void getExtsysId(ASAP_EXTSYS_ID& extsysId) const;
CS_INT getEventStatus(void) const;
void setWoId (const WORK_ORD_NUM woId);
void setEventUnId(const CS_INT eventUnId);
void setExtsysId(const ASAP_EXTSYS_ID extsysId);
void setEventStatus(const CS_INT eventStatus);

};
```

Public methods

The following are the public methods.

getWoId

Returns the work order identification.

Syntax:

```
void getWoId(WORK_ORD_NUM& woId) const;
```

getEventUnId

Returns the event unit ID.

Syntax:

```
CS_INT getEventUnId(void) const;
```

getExtsysId

Returns the external system ID.

Syntax:

```
void getExtsysId(ASAP_EXTSYS_ID& extsysId) const;
```

getEventStatus

Returns the event status.

Syntax:

```
CS_INT getEventStatus(void) const;
```

setWoId

Sets the work order identification.

Syntax:

```
void setWoId (const WORK_ORD_NUM woId);
```

setEventUnId

Sets the event unit ID.

Syntax:

```
void setEventUnId(const CS_INT eventUnId);
```

setExtsysId

Sets the external system ID.

Syntax:

```
void setExtsysId(const ASAP_EXTSYS_ID extsysId);
```

setEventStatus

Sets the event status.

Syntax:

```
void setEventStatus(const CS_INT eventStatus);
```

SRP_WoCompleteEvent class

This class delivers the WO_COMPLETE event information to the completion event handler. It inherits from the **SRP_Event** class.

Synopsis

```
class SRP_WoCompleteEvent : public SRP_Event
{
public:

SRP_WoCompleteEvent(void);
SRP_WoCompleteEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_REVISIONS_FLAG
```

```
revFlag, ASAP_EXCEPTIONS_FLAG except);

void getRevFlag(ASAP_REVISIONS_FLAG& revFlag) const;
void getExcept(ASAP_EXCEPTIONS_FLAG& except) const;
void setRevFlag(const ASAP_REVISIONS_FLAG revFlag);
void setExcept(const ASAP_EXCEPTIONS_FLAG except);

};
```

Public methods

The following are the public methods.

getRevFlag

Returns the revision flag.

Syntax:

```
void getRevFlag(ASAP_REVISIONS_FLAG& revFlag) const;
```

getExcept

Returns the exception flag.

Syntax:

```
void getExcept(ASAP_EXCEPTIONS_FLAG& except) const;
```

setRevFlag

Sets the revision flag.

Syntax:

```
void setRevFlag(const ASAP_REVISIONS_FLAG revFlag);
```

setExcept

Sets the exception flag.

Syntax:

```
void setExcept(const ASAP_EXCEPTIONS_FLAG except);
```

SRP_WoFailureEvent class

The **SRP_WoFailureEvent** class delivers the WO_FAILURE event information to the failure event handler. This class inherits from the **SRP_Event** class.

Synopsis

```
class SRP_WoFailureEvent : public SRP_Event
{
public:

SRP_WoFailureEvent(void);
SRP_WoFailureEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
```

```
ASAP_EXTSYS_ID extSysId, ASAP_CSDL_SEQ_NO csdlSeqNo,
ASAP_CSDL_ID csdlId);
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
ASAP_CSDL_ID getCsdlId(void) const;
void setCsdlSeqNo(const ASAP_CSDL_SEQ_NO csdlSeqNo);
void setCsdlId(const ASAP_CSDL_ID csdlId);

};
```

Public methods

The following are the public methods.

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
```

getCsdlId

Returns the CSDL ID.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_CSDL_SEQ_NO csdlSeqNo);
```

setCsdlId

Sets the CSDL ID.

Syntax:

```
void setCsdlId(const ASAP_CSDL_ID csdlId);
```

SRP_SoftErrorEvent class

The **SRP_SoftErrorEvent** class delivers the SOFT_ERROR event information to the soft error event handler. This class inherits from the **SRP_Event** class.

Synopsis

```
class SRP_SoftErrorEvent : public SRP_Event
{
public:

SRP_SoftErrorEvent(void);
SRP_SoftErrorEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
```

```
ASAP_EXTSYS_ID extSysId, ASAP_CSDL_SEQ_NO csdlSeqNo,
ASAP_CSDL_ID csdlId);
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
ASAP_CSDL_ID getCsdlId(void) const;
void setCsdlSeqNo(const ASAP_CSDL_SEQ_NO csdlSeqNo);
void setCsdlId(const ASAP_CSDL_ID csdlId);

};
```

Public methods

The following are the public methods.

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
```

getCsdlId

Returns the CSDL ID.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_CSDL_SEQ_NO csdlSeqNo);
```

setCsdlId

Sets the CSDL ID.

Syntax:

```
void setCsdlId(const ASAP_CSDL_ID csdlId);
```

SRP_WoEstimateEvent class

The **SRP_WoEstimateEvent** class delivers the WO_ESTIMATE event information to the work order estimate event handler.

Synopsis

```
class SRP_WoEstimateEvent: public SRP_Event
{
public:

SRP_WoEstimateEvent(void);
SRP_WoEstimateEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
```

```
ASAP_EXTSYS_ID extSysId, ASAP_ESTIMATE estimate, ASAP_WO_MISC misc);
ASAP_ESTIMATE getEstimate(void) const;
void getMisc(ASAP_WO_MISC& misc) const;
void setEstimate(const ASAP_ESTIMATE estimate);
void setMisc(const ASAP_WO_MISC misc);

};
```

Public methods

The following are the public methods.

getEstimate

Returns the estimation.

Syntax:

```
ASAP_ESTIMATE getEstimate(void) const;
```

getMisc

Returns miscellaneous information.

Syntax:

```
void getMisc(ASAP_WO_MISC& misc) const;
```

setEstimate

Sets the estimation.

Syntax:

```
void setEstimate(const ASAP_ESTIMATE estimate);
```

setMisc

Sets the miscellaneous information.

Syntax:

```
void setMisc(const ASAP_WO_MISC misc);
```

SRP_WoStartupEvent class

The **SRP_WoStartupEvent** class delivers the WO_STARTUP event information to the work order startup handler.

Synopsis

```
class SRP_WoStartupEvent: public SRP_Event
{
public:

SRP_WoStartupEvent(void);
SRP_WoStartupEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId);
```

```
};
```

SRP_WoRollbackEvent class

The **SRP_RollbackEvent** class is used to deliver the WO_ROLLBACK event information to the work order roll-back event handler.

To inplement rollback of completed ASDLs, you must configure the WO_ROLLBACK, ROLLBACK_REQ, and IGNORE_ROLLBACK variables.

Synopsis

```
class SRP_WoRollbackEvent: public SRP_Event
{
public:

SRP_WoRollbackEvent(void);
SRP_WoRollbackEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId);

};
```

SRP_NEUnknownEvent class

The **SRP_NEUnknownEvent** class delivers the NE_UNKNOWN event information to the NE unknown event handler.

Synopsis

```
class SRP_NEUnknownEvent : public SRP_Event
{
public:

SRP_NEUnknownEvent(void);
SRP_NEUnknownEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_CSDL_SEQ_NO csdlSeqNo,
ASAP_CSDL_ID csdlId, ASAP_CLLI machClli);
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
ASAP_CSDL_ID getCsdlId(void) const;
void getMachClli(ASAP_CLLI& machClli) const;
void setCsdlSeqNo(const ASAP_CSDL_SEQ_NO csdlSeqNo);
void setCsdlId(const ASAP_CSDL_ID csdlId);
void setMachClli(const ASAP_CLLI machClli);

};
```

Public methods

The following are the public methods.

getCsdlSeqNo

Returns the order sequence of the CSDL command within the work order.

Syntax:

```
ASAP_CSDL_SEQ_NO getCsdlSeqNo(void) const;
```

getCsdlId

Returns the CSDL ID. The CSDL ID uniquely identifies the CSDL command to be provisioned.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

getMachClli

Returns the machine CLLI. The machine CLLI identifies the NE that is not known to SARM.

Syntax:

```
void getMachClli(ASAP_CLLI& machClli) const;
```

SRP_WoBlockEvent class

The **SRP_WoBlockEvent** class delivers the WO_BLOCKED event information to the work order block event handler.

Synopsis

```
class SRP_WoBlockEvent : public SRP_Event
{
public:

SRP_WoBlockEvent(void);
SRP_WoBlockEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_WO_BLOCK_REASON
m_reason);
void getReason(ASAP_WO_BLOCK_REASON& reason) const;
void setReason(const ASAP_WO_BLOCK_REASON reason);

};
```

Public methods

The following are the public methods.

getReason

Returns the reason why provisioning of the work order has been blocked.

Syntax:

```
void getReason(ASAP_WO_BLOCK_REASON& reason) const;
```

setReason

Sets the reason why provisioning of the work order can be blocked.

Syntax:

```
void setReason(const ASAP_WO_BLOCK_REASON reason);
```

SRP_WoTimeOutEvent class

The **SRP_WoTimeOutEvent** class delivers the WO_TIMEOUT event information to the work order timeout event handler.

Synopsis

```
class SRP_WoTimeOutEvent: public SRP_Event
{
public:

SRP_WoTimeOutEvent(void);
SRP_WoTimeOutEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_WO_TIMEOUT_STATUS status);
ASAP_WO_TIMEOUT_STATUS getStatus(void) const;
void setStatus(const ASAP_WO_TIMEOUT_STATUS status);

};
```

Public methods

The following are the public methods.

getStatus

Returns the status of the work order. Possible values include:

(ASAP_TIMEOUT_EXECUTING, ASAP_TIMEOUT_FAIL, ...) These are defined in the “asap_core.h".

Syntax:

```
ASAP_WO_TIMEOUT_STATUS getStatus(void) const;
```

setStatus

```
void setStatus(const ASAP_WO_TIMEOUT_STATUS status);
```

SRP_NEAvailEvent class

The **SRP_NEAvailEvent** class delivers the NE_AVAIL event information to the NE available event handler.

Synopsis

```
class SRP_NEAvailEvent : public SRP_Event
{
public:

SRP_NEAvailEvent(void);
SRP_NEAvailEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_CLLI hostClli);
void getHostClli(ASAP_CLLI& hostClli) const;
void setHostClli(const ASAP_CLLI hostClli);
```

```
};
```

Public methods

The following are the public methods.

getHostClli

```
void getHostClli(ASAP_CLLI& hostClli) const;
```

Syntax:

Returns the host CLLI.

setHostClli

```
void setHostClli(const ASAP_CLLI hostClli);
```

SRP_NEUnAvailEvent class

The **SRP_NeUnAvailEvent** class delivers the NE_UNAVAIL event information to the NE unavailable event handler.

Synopsis

```
class SRP_NEUnavailEvent : public SRP_Event
{
public:

SRP_NEUnavailEvent(void);
SRP_NEUnavailEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, ASAP_CLLI hostClli);

void getHostClli(ASAP_CLLI& hostClli) const;
void setHostClli(const ASAP_CLLI hostClli);

};
```

Public methods

The following are the public methods.

getHostClli

Returns the host CLLI.

Syntax:

```
void getHostClli(ASAP_CLLI& hostClli) const;
```

setHostClli

```
void setHostClli(const ASAP_CLLI hostClli);
```

SRP_WoAcceptEvent class

The **SRP_WoAcceptEvent** class is used to deliver the WO_ACCEPT event information to the work order accepted event handler.

Synopsis

```
class SRP_WoAcceptEvent : public SRP_Event
{
public:

SRP_WoAcceptEvent(void);
SRP_WoAcceptEvent(WORK_ORD_NUM woId, CS_INT eventUnId,
ASAP_EXTSYS_ID extSysId, CS_INT newWoStat, CS_INT
oldWoStat, CS_INT status);
CS_INT getNewWoStat(void) const;
CS_INT getOldWoStat(void) const;
CS_INT getStatus(void) const;
void setNewWoStat(CS_INT newWoStat);
void setOldWoStat(CS_INT oldWoStat);
void setStatus(CS_INT status);

};
```

Public methods

The following are the public methods.

getNewWoStat

Returns the new work order status.

Syntax:

```
CS_INT getNewWoStat(void) const;
```

getOldWoStat

Returns the old work order status.

Syntax:

```
CS_INT getOldWoStat(void) const;
```

getStatus

Returns the event status.

Syntax:

```
CS_INT getStatus(void) const;
```

setNewWoStat

Returns the new work order status.

Syntax:

```
CS_INT setNewWoStat(void) const;
```

setOldWoStat

Returns the old work order status.

Syntax:

```
CS_INT setOldWoStat(void) const;
```

setStatus

Returns the event status.

Syntax:

```
CS_INT setStatus(void) const;
```

ASC_RetrieveInfo class

This class is the base class of all other information classes described in this section.

Synopsis

```
class ASC_RetrieveInfo
{
public:

    ASC_RetrieveInfo(void) {}
    ~ASC_RetrieveInfo(void) {}
    void getWoId(ASAP_WO_ID& woId) const
    { strcpy(woId, m_woId); }
    VOIDPTR getDataPtr(void) const
    { return m_dataPtr; }
    void setWoId(const ASAP_WO_ID woId)
    {
    ::memset(m_woId, '\0', sizeof(ASAP_WO_ID));
    ::strncpy(m_woId, woId, sizeof(ASAP_WO_ID)-1);
    }
    void setDataPtr(const VOIDPTR dataPtr)     { m_dataPtr = dataPtr; }

private:
    ASAP_WO_ID m_woId;
    VOIDPTR m_dataPtr;
};
```

Constructor

Constructor of the class.

Syntax:

```
ASC_RetrieveInfo(void);
```

Public methods

The following are the public methods.

getWoId

Returns the work order ID.

Syntax:

```
void getWoId(ASAP_WO_ID& woId) const;
```

setWoId

Sets the work order ID. Ensure the work order ID does not exceed the length of the **ASAP_WO_ID** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setWoId(const ASAP_WO_ID woId)
{
::memset(m_woId, '\0', sizeof(ASAP_WO_ID));::strncpy(m_woId, woId,
sizeof(ASAP_WO_ID)-1);
}
```

getDataPtr

Returns the data pointer. Currently not used by C++ SRP API.

Syntax:

```
VOIDPTR getDataPtr(void) const
     { return m_dataPtr; }
```

setDataPtr

Sets the data pointer. Currently not used by C++ SRP API.

Syntax:

```
void setDataPtr(const VOIDPTR dataPtr)
     { m_dataPtr = dataPtr; }
```

ASC_CsdlListInfo class

This class defines the information objects for CSDL list retrieval information. The CSDL list information is retrieved from the SARM database row by row and parsed into the **ASC_CsdlListInfo** objects. All objects are inserted into an **ASC_RetrieveInfoSet** container object, which is then passed to the handler that you define.

Synopsis

```
class ASC_CsdlListInfo : public ASC_RetrieveInfo
{
public:
```

```
ASC_CsdlListInfo(void)
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
ASAP_STAT getCsdlStat(void) const;
ASAP_SEQ_NO getCsdlSeqNo(void) const;
ASAP_CSDL_ID getCsdlId(void) const;
ASAP_CSDL_ESTIM getCsdlEst(void) const;
void getCsdlDesc(ASAP_CSDL_DESC& csdlDesc) const;
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
void setCsdlStat(const ASAP_STAT csdlStat);
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
void setCsdlId(const ASAP_CSDL_ID csdlId);
void setCsdlEst(const ASAP_CSDL_ESTIM csdlEst);
void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);

};
```

Constructor

Constructor of the class.

Syntax:

```
ASC_CsdlListInfo(void);
```

Public methods

The following are the public methods.

getCsdlCmd

Returns the CSDL command.

Syntax:

```
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
```

setCsdlCmd

Sets the CSDL command. The CSDL command length must not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
```

getCsdlStat

Returns the CSDL state. Possible values include ASAP_CSDL_INITIAL, ASAP_CSDL_HELD, ASAP_CSDL_MANUAL_TRAN, ASAP_CSDL_TRAN_ERR.

Syntax:

```
ASAP_STAT getCsdlStat(void) const;
```

setCsdlStat

Sets the CSDL state. Possible values include ASAP_CSDL_INITIAL, ASAP_CSDL_HELD, ASAP_CSDL_MANUAL_TRAN, ASAP_CSDL_TRAN_ERR.

Syntax:

```
void setCsdlStat(const ASAP_STAT csdlStat);
```

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_SEQ_NO getCsdlSeqNo(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
```

getCsdlId

Returns the CSDL ID.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

setCsdlId

Sets the CSDL ID.

Syntax:

```
void setCsdlId(const ASAP_CSDL_ID csdlId);
```

getCsdlEst

Returns the CSDL estimation.

Syntax:

```
ASAP_CSDL_ESTIM getCsdlEst(void) const;
```

setCsdlEst

Sets the CSDL estimation.

Syntax:

```
void setCsdlEst(const ASAP_CSDL_ESTIM csdlEst);
```

getCsdlDesc

Returns the CSDL description. The length of the description must not exceed the size of **ASAP_CSDL_DESC** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getCsdlDesc(ASAP_CSDL_DESC& csdlDesc) const;
```

setCsdlDesc

Sets the CSDL description. The length of the description should not exceed the size of **ASAP_CSDL_DESC** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);
```

ASC_CsdlLogInfo class

The **ASC_CsdlLogInfo** class defines the information objects for CSDL log retrieval information. The CSDL log information is retrieved from the SARM database row by row, and parsed into the **ASC_CsdlLogInfo** objects. All objects will be inserted to an ASC_RetrieveInfoSet container object, which will in turn be passed to the user-defined handler.

Synopsis

```
class ASC_CsdlLogInfo : public ASC_RetrieveInfo
{
public:

ASC_CsdlLogInfo(void);

ASAP_DTS getDateTm(void) const;
void getEventType(SRQ_EVENT_TYPE& evtBuf) const;
void getEventText(SRQ_EVENT_TEXT& evtLineBuf) const;
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
ASAP_SEQ_NO getCsdlSeqNo(void) const;
void getHostClli(ASAP_CLLI& hostClli) const;
void setDateTm(const ASAP_DTS dateTm);
void setEventType(const SRQ_EVENT_TYPE evtBuf);
void setEventText(const SRQ_EVENT_TEXT evtLineBuf);
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
void setHostClli(const ASAP_CLLI hostClli);

}
```

Public methods

Constructor of the class.

See also ASC_RetrieveRequest, ASC_RetrieveInfo, and ASC_RetrieveInfoSet.

Syntax:

```
ASC_CsdlLogInfo(void);
```

getDateTm

Returns the date and time.

Syntax:

```
ASAP_DTS getDateTm(void) const;
```

setDateTm

Sets the date and time.

Syntax:

```
void setDateTm(const ASAP_DTS dateTm);
```

getEventType

Returns the event types. Ensure the type name does not exceed the length of the **SRQ_EVENT_TYPE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getEventType(SRQ_EVENT_TYPE& evtBuf) const;
```

setEventType

Sets the event types. Ensure the type name does not exceed the length of the **SRQ_EVENT_TYPE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setEventType(const SRQ_EVENT_TYPE evtBuf);
```

getEventText

Returns the event text. Ensure the event text does not exceed the length of the **SRQ_EVENT_TEXT** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getEventText(SRQ_EVENT_TEXT& evtLineBuf) const;
```

setEventText

Sets the event text. Ensure the event text does not exceed the length of the **SRQ_EVENT_TEXT** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setEventText(const SRQ_EVENT_TEXT evtLineBuf);
```

getCsdlCmd

Returns the CSDL command. Ensure the CSDL command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
```

setCsdlCmd

Sets the CSDL command. Ensure the CSDL command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
```

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_SEQ_NO getCsdlSeqNo(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
```

getHostClli

Returns the host_clli name. Ensure the host_clli name does not exceed the length of the **ASAP_CLLI** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getHostClli(ASAP_CLLI& hostClli) const;
```

setHostClli

Sets the host_clli name. Ensure the host_clli name does not exceed the length of the **ASAP_CLLI** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setHostClli(const ASAP_CLLI hostClli);
```

ASC_WoLogInfo class

This class defines the information objects for work order log retrieval information. The work order log information is retrieved from the SARM database row by row, and parsed into the **ASC_WoLogInfo** objects. All objects will be inserted to an ASC_RetrieveInfoSet container object, which will in turn be passed to the user-defined handler.

Synopsis

```
class ASC_WoLogInfo : public ASC_RetrieveInfo
{
public:
```

```
ASC_WoLogInfo(void);
ASAP_DTS getDateTm(void) const;
void getEventType(SRQ_EVENT_TYPE& evtBuf) const;
void getEventText(SRQ_EVENT_TEXT& evtLineBuf) const;
ASAP_SEQ_NO getCsdlSeqNo(void) const;
void getHostClli(ASAP_CLLI& hostClli) const;
void setDateTm(const ASAP_DTS dateTm);
void setEventType(const SRQ_EVENT_TYPE evtBuf);
void setEventText(const SRQ_EVENT_TEXT evtLineBuf);
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
void setHostClli(const ASAP_CLLI hostClli);

}
```

Public methods

Constructor of the class.

See also ASC_RetrieveRequest, ASC_RetrieveInfo, and ASC_RetrieveInfoSet.

Syntax:

```
ASC_WoLogInfo(void);
```

getDateTm

Returns the date and time.

Syntax:

```
ASAP_DTS getDateTm(void) const;
```

setDateTm

Sets the date and time.

Syntax:

```
void setDateTm(const ASAP_DTS dateTm);
```

getEventType

Returns the event type name. Ensure the type name does not exceed the length of the **SRQ_EVENT_TYPE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getEventType(SRQ_EVENT_TYPE& evtBuf) const;
```

setEventType

Sets the event type name. Ensure the type name does not exceed the length of the **SRQ_EVENT_TYPE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setEventType(const SRQ_EVENT_TYPE evtBuf);
```

getEventText

Returns the event text. Ensure the event text does not exceed the length of the **SRQ_EVENT_TEXT** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getEventText(SRQ_EVENT_TEXT& evtLineBuf) const;
```

setEventText

Sets the event text. Ensure the event text does not exceed the length of the **SRQ_EVENT_TEXT** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setEventText(const SRQ_EVENT_TEXT evtLineBuf);
```

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_SEQ_NO getCsdlSeqNo(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
```

getHostClli

Returns the host_clli name. Ensure the host_clli name does not exceed the length of the **ASAP_CLLI** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getHostClli(ASAP_CLLI& hostClli) const;
```

setHostClli

Sets the host_clli name. Ensure the host_clli name does not exceed the length of the ASAP_CLLI data type. Otherwise, the work order will be rejected.

Syntax:

```
void setHostClli(const ASAP_CLLI hostClli);
```

ASC_WoParamInfo class

This class defines the information objects for the retrieval of work order parameter information. The work order parameter information is retrieved from the SARM database row by row, and parsed into the **ASC_WoParamInfo** objects on a one-row-

one-object basis. All objects will be inserted to an **ASC_RetrieveInfoSet** container object, which will in turn be passed to the user-defined handler.

Synopsis

```
class ASC_WoParamInfo : public ASC_RetrieveInfo
{
public:

ASC_WoParamInfo(void);
void getParmGrp(ASAP_PARM_GRP& parmGrp) const;
void getParmLbl(ASAP_CSDL_LABEL& parmLbl) const;
void getParmVlu(ASAP_CSDL_VALUE& parmVlu) const;
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
ASAP_SEQ_NO getCsdlSeqNo(void) const;
ASAP_CSDL_ID getCsdlId(void) const;
void setParmGrp(const ASAP_PARM_GRP parmGrp);
void setParmLbl (const ASAP_CSDL_LABEL parmLbl);
void setparmVlu(const ASAP_CSDL_VALUE parmVlu);
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
void setCsdlId(ASAP_CSDL_ID csdlId);

};
```

Public methods

Constructor of the class.

See also ASC_RetrieveRequest, ASC_RetrieveInfo, and ASC_RetrieveInfoSet.

Syntax:

```
ASC_WoParamInfo(void);
```

getParmGrp

Returns the parameter group name. Ensure the group name does not exceed the length of the **ASAP_PARM_GRP** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getParmGrp(ASAP_PARM_GRP& parmGrp) const;
```

setParmGrp

Sets the parameter group name. Ensure the group name does not exceed the length of the **ASAP_PARM_GRP** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setParmGrp(const ASAP_PARM_GRP parmGrp);
```

getParmLbl

Returns the parameter label name. Ensure the label name does not exceed the length of the **ASAP_CSDL_LABEL** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getParmLbl(ASAP_CSDL_LABEL& parmLbl) const;
```

setparmLbl

Sets the parameter label name. Ensure the label name does not exceed the length of the **ASAP_CSDL_LABEL** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setparmLbl(const ASAP_CSDL_LABEL parmLbl);
```

getParmVlu

Returns the parameter value. Ensure the value does not exceed the length of the **ASAP_CSDL_VALUE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getParmVlu(ASAP_CSDL_VALUE& parmVlu) const;
```

setparmVlu

Sets the parameter value. Ensure the value does not exceed the length of the **ASAP_CSDL_VALUE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setparmVlu(const ASAP_CSDL_VALUE parmVlu);
```

getCsdlCmd

Returns the CSDL command. Ensure the command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
```

setCsdlCmd

Sets the CSDL command. Ensure the command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
```

getCsdlSeqNo

Returns the CSDL sequence number.

Syntax:

```
ASAP_SEQ_NO getCsdlSeqNo(void) const;
```

setCsdlSeqNo

Sets the CSDL sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
```

getCsdlId

Returns the CSDL ID.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

setCsdlId

Sets the CSDL ID.

Syntax:

```
void setCsdlId(ASAP_CSDL_ID csdlId);
```

ASC_WoRevInfo class

This class defines the information objects for the retrieval information of work order revisions. The work order revision information is retrieved from the SARM database row by row, and parsed into the **ASC_WoRevInfo** objects on a one-row-one-object basis. All objects will be inserted to an **ASC_RetrieveInfoSet** container object, which will in turn be passed to the user-defined handler.

Synopsis

```
class ASC_WoRevInfo : public ASC_RetrieveInfo
{
public:

ASC_WoRevInfo(void);
CS_CHAR getRevFlag(void) const;
void getLabel(ASAP_CSDL_LABEL& label) const;
void getValue(ASAP_CSDL_VALUE& value) const;
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
void getCsdlDesc(ASAP_CSDL_DESC& csdlDesc) const;
ASAP_SEQ_NO getCsdlSeqNo(void) const;
ASAP_SEQ_NO getParmSeqNo(void) const;
void setRevFlag(const CS_CHAR revFlag);
void setLabel(const ASAP_CSDL_LABEL label);
void setValue(const ASAP_CSDL_VALUE value);
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
void setParmSeqNo(const ASAP_SEQ_NO parmSeqNo);

};
```

Public methods

Constructor of the class.

See also ASC_RetrieveRequest, ASC_RetrieveInfo, and ASC_RetrieveInfoSet.

Syntax:

```
ASC_WoRevInfo(void);
```

getRevFlag

Returns the revision flag. The revision flag indicates whether a work order CSDL has been revised, possible values include **ASAP_WO_REVISIONS**, **ASAP_WO_NO_REVISIONS**.

Syntax:

```
CS_CHAR getRevFlag(void) const;
```

setRevFlag

Sets the revision flag. The revision flag indicates whether a work order CSDL has been revised, possible values include **ASAP_WO_REVISIONS**, **ASAP_WO_NO_REVISIONS**.

Syntax:

```
void setRevFlag(const CS_CHAR revFlag);
```

getLabel

Returns the parameter label. Ensure the label name does not exceed the length of the **ASAP_CSDL_LABEL** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getLabel(ASAP_CSDL_LABEL& label) const;
```

setLabel

Sets the parameter label. Ensure the label name does not exceed the length of the **ASAP_CSDL_LABEL** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setLabel(const ASAP_CSDL_LABEL label);
```

getValue

Returns the parameter value. Ensure the value does not exceed the length of the **ASAP_CSDL_VALUE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getValue(ASAP_CSDL_VALUE& value) const;
```

setValue

Sets the parameter value. Ensure the value does not exceed the length of the **ASAP_CSDL_VALUE** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setValue(const ASAP_CSDL_VALUE value);
```

getCsdlCmd

Returns the CSDL command. Ensure the command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getCsdlCmd(ASAP_CSDL_CMD& csdlCmd) const;
```

setCsdlCmd

Sets the CSDL command. Ensure the command does not exceed the length of the **ASAP_CSDL_CMD** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlCmd(const ASAP_CSDL_CMD csdlCmd);
```

getCsdlDesc

Returns a brief CSDL description. Ensure the description does not exceed the length of the **ASAP_CSDL_DESC** data type. Otherwise, the work order will be rejected.

Syntax:

```
void getCsdlDesc(ASAP_CSDL_DESC& csdlDesc) const;
```

setCsdlDesc

Sets a brief CSDL description. Ensure the description does not exceed the length of the **ASAP_CSDL_DESC** data type. Otherwise, the work order will be rejected.

Syntax:

```
void setCsdlDesc(const ASAP_CSDL_DESC csdlDesc);
```

getCsdlSeqNo

Returns the CSDL command sequence number.

Syntax:

```
ASAP_SEQ_NO getCsdlSeqNo(void) const;
```

setCsdlSeqNo

Sets the CSDL command sequence number.

Syntax:

```
void setCsdlSeqNo(const ASAP_SEQ_NO csdlSeqNo);
```

getParmSeqNo

Get/set the parameter sequence number.

Syntax:

```
ASAP_SEQ_NO getParmSeqNo(void) const;
```

setParmSeqNo

Get/set the parameter sequence number.

Syntax:

```
void setParmSeqNo(const ASAP_SEQ_NO parmSeqNo);
```

ASC_RetrieveInfoSet class

This class defines the container for the information objects described above. An **ASC_RetrieveInfoSet** object, which possibly contains zero to multiple information objects, is passed to the user-defined handler for a particular type of retrieval when the retrieval is done. You must retrieve the information objects from the **ASC_RetrieveInfoSet** object,and process them before returning from the handler.

The C++ SRP API library will delete the **ASC_RetrieveInfoSet** object and all the information objects it contains, upon the termination of the user-defined handler. Make copies if you want to keep the information beyond the lifetime of the handler.

Synopsis

```
class ASC_RetrieveInfoSet
{
public:
    ASC_RetrieveInfoSet(void)
    : m_iter(retInfoList.begin()) {}
    ~ASC_RetrieveInfoSet(void);
    void goToHead(void);
    int itemCount(void);
    void *goToNext(void);
    void *removeNext(void);
    void *insertItem(void *c);

private:
    std::list<ASC_RetrieveInfo *> retInfoList;
    std::list<ASC_RetrieveInfo *>::iterator m_iter;
    int current_position;
};
```

Public methods

Constructor of the class.

See also ASC_RetrieveRequest, ASC_CsdlListInfo, ASC_CsdlLogInfo, ASC_WoLogInfo, ASC_WoParamInfo, ASC_WoRevInfo, and ASC_RetrieveInfo.

Syntax:

```
ASC_RetrieveInfoSet(void);
```

goToHead

This method resets the retrieval pointer to the starting status.

```
void goToHead(void);
```

itemCount

This method returns the number of items in the container.

Syntax:

```
int itemCount(void);
```

goToNext

This method moves the retrieval pointer to the next item and returns it. The returned item is not removed from the list. NULL will be returned if the end of the list is reached.

Syntax:

```
void *goToNext(void);
```

removeNext

This method removes the next item from the list and returns it. If the end of the list is reached, it will return NULL. If you remove an information object from the container, you must delete this object.

Syntax:

```
void *removeNext(void);
```

insertItem

This method inserts an item into the container. If the equivalent is already in, the new one will not be inserted and the old one is returned.

Syntax:

```
void *insertItem(void *c);
```

ASC_RetrieveRequest class

This class is used to construct retrieval requests to the SARM databases. Whenever you need to retrieve the persistent data (for example, work order logs, revisions, parameters, and/or CSDL logs, lists, parameters, and/or SRP event handler listening port) or rebuild a work order from the SARM database, a request object of this class type should be created.

The class provides five default (do nothing) processing functions for retrieval types of CSDL list, CSDL logs, work order parameters, work order revisions, and work order logs, respectively.

Create a subclass from this class and override these processing functions if you would like to handle the retrieval results with your own logic.

There are no virtual processing functions defined for rebuilding work order objects and retrieving the SRP event handler listen port, because a request object directly populates the work order object passed to it, and copies the port number (string) to the user-supplied buffer.

Synopsis

```
class ASC_WoRetrieveRequest : public Diagnosis
```

```
// DESC: This class is used to construct a retrieval request and then
// fire the request down.

{
public:

// For WO_REVS, CSDL_LIST, REBUILD_WO.
ASC_WoRetrieveRequest(const ASAP_WO_ID woId,
const CS_INT retrieveType);

// For WO_PARAM.
ASC_WoRetrieveRequest(const ASAP_WO_ID woId,
const ASAP_CSDL_LABEL label, const ASAP_PARM_GRP group);

// For WO_LOG.
ASC_WoRetrieveRequest(const ASAP_WO_ID woId,
const SRQ_EVENT_TYPE srqEvt, const CS_BOOL neRespLineByLine);

// For CSDL_LOG.
ASC_WoRetrieveRequest(const ASAP_WO_ID woId,
const SRQ_EVENT_TYPE logType, const ASAP_CSDL_ID csdlId,
const CS_BOOL neRespLineByLine);

// For SRP PORT Retrieval.
ASC_WoRetrieveRequest(void) {}

// Following functions are used by ASC_WoRetrieveInterface.

CS_INT getRetrieveType(void) const;
void getWoId(ASAP_WO_ID& woId) const;
void getLogType(SRQ_EVENT_TYPE& logType) const;
ASAP_CSDL_ID getCsdlId(void) const;
CS_BOOL getNeRespLineByLine(void) const;
void getSrqEvt(SRQ_EVENT_TYPE& srqEvt) const;
void getLabel(ASAP_CSDL_LABEL& label) const;
void getGroup(ASAP_PARM_GRP& group) const;
CS_RETCODE retrieve(void);
CS_RETCODE rebuildWo(SRP_WO *wo);
CS_RETCODE getMyPort(char *srpId, SRP_PORT_NUM& port) const;

virtual void csdlListProcessFn(ASC_RetrieveInfoSet *) {}
virtual void csdlLogProcessFn(ASC_RetrieveInfoSet *) {}
virtual void woLogProcessFn(ASC_RetrieveInfoSet *) {}
virtual void woParamProcessFn(ASC_RetrieveInfoSet *) {}
virtual void woRevProcessFn(ASC_RetrieveInfoSet *) {}

};
```

Public methods

Constructor for types of work order revision, CSDL list, and rebuilding work order.

See also ASC_RetrieveInfoSet, ASC_CsdlListInfo, ASC_CsdlLogInfo, ASC_WoLogInfo, ASC_WoParamInfo, ASC_WoRevInfo, and ASC_RetrieveInfo.

```
ASC_WoRetrieveRequest(const ASAP_WO_ID woId, const CS_INT retrieveType);
```

Constructor for work order parameter retrieval type.

```
ASC_WoRetrieveRequest(const ASAP_WO_ID woId, const ASAP_CSDL_LABEL label, const
ASAP_PARM_GRP group);
```

Constructor for work order log retrieval type.

```
ASC_WoRetrieveRequest(const ASAP_WO_ID woId, const SRQ_EVENT_TYPE srqEvt, const
CS_BOOL neRespLineByLine);
```

Constructor for CSDL log retrieval type.

```
ASC_WoRetrieveRequest(const ASAP_WO_ID woId, const SRQ_EVENT_TYPE logType, const
ASAP_CSDL_ID csdlId, const CS_BOOL neRespLineByLine);
```

Constructor for retrieving SRP listen-port.

```
ASC_WoRetrieveRequest(void);
```

getRetrieveType

Get the retrieval type of the object.

Syntax:

```
CS_INT getRetrieveType(void) const;
```

getUserData

Get user-defined data from the object.

Syntax:

```
VOIDPTR getUserData(void) const;
```

getWoId

Get the work order ID from the object.

Syntax:

```
void getWoId(ASAP_WO_ID& woId) const;
```

getLogType

Get the log type of the request.

Syntax:

```
void getLogType(SRQ_EVENT_TYPE& logType) const;
```

getCsdlId

Get the CSDL ID recorded by the object.

Syntax:

```
ASAP_CSDL_ID getCsdlId(void) const;
```

getNeRespLineByLine

Get the flag for whether one-object-one-line mode should be used for CSDL or work order log retrievals.

Syntax:

```
CS_BOOL getNeRespLineByLine(void) const;
```

getSrqEvt

Get the retrieval event type for the request.

Syntax:

```
void getSrqEvt(SRQ_EVENT_TYPE& srqEvt) const;
```

getLabel

Get the parameter label.

Syntax:

```
void getLabel(ASAP_CSDL_LABEL& label) const;
```

getGroup

Get the group name for the work order parameter retrieval.

Syntax:

```
void getGroup(ASAP_PARM_GRP& group) const;
```

retrieve

Start up the retrieval (of type WO_REVS, CSDL_LIST, CSDL_LOG, WO_LOG, WO_PARAM) defined by the object. The function will return CS_SUCCEEDon successfully completing the retrieval; otherwise it will return CS_FAIL.

Syntax:

```
CS_RETCODE retrieve(void);
```

rebuildWo

Start up to rebuild a work order object. The input argument is a pointer to the work order skeketon object. Upon successful completion, the function returns CS_SUCCEED. The work order object has been populated with all the attributes, global parameters, and CSDLs and their parameters.

Syntax:

```
CS_RETCODE rebuildWo(SRP_WO *wo);
```

getMyPort

Get the SRP event handler's listen port number from the SARM database. If successful, CS_SUCCEED will be returned with the argument port populated with the port number (string). If the retrieval is not successful, CS_FAIL will be returned and port will have a value of “0".

Syntax:

```
CS_RETCODE getMyPort(char *srpId, SRP_PORT_NUM& port) const; virtual void
csdlListProcessFn(ASC_RetrieveInfoSet *); virtual void
csdlLogProcessFn(ASC_RetrieveInfoSet *); virtual void
woLogProcessFn(ASC_RetrieveInfoSet *); virtual void
woParamProcessFn(ASC_RetrieveInfoSet *); virtual void
woRevProcessFn(ASC_RetrieveInfoSet *);
```

Retrieval results in processing functions for CSDL list, CSDL log, work order log, work order parameter, and work order revision retrievals, respectively. No action is defined in any of these functions. You should subclass this class to provide your own processing functions.

You should not delete the **ASC_RetrieveInfoSet** object passed to a processing function and any information objects contained in the **ASC_RetrieveInfoSet** object.

> **Note:**
>
> You should make your own copy of this information if you want to keep the information stored in an object beyond the lifetime of the processing function.

5
Downstream Interfaces

This chapter describes the following downstream interfaces:

- NEP library
- Protocol-specific libraries
- External device driver
- Action functions

NEP library

If the customized NEP has links to the Communication library, you will only need to define the CMD_com_init() function, since libasccomm has the CMP_connect_port() and CMD_disconnect_port() API calls.

This chapter defines the base requirements of the command processor. These base requirements include the mandatory functions required by the NEP core system to generate a customized NEP server. The required functions are listed in this chapter with their appropriate prototypes and brief descriptions.

The NEP Library contains the following functions:

- **ASC_loadCommParams:** Returns the list of communication parameters for the specified device type, host, and device.
- **CMD_comm_init:** Initializes the communications interface library.
- **CMD_connect_port:** Opens the connection to the device specified by the port information structure.
- **CMD_disconnect_port:** Closes the connection to the device specified by the port information structure.

NEP library functions

This section details all the functions (in alphabetical order) in the NEP Library.

ASC_loadCommParams

This function returns the list of communication parameters for the specified device type, host, and device.

Syntax:

```
CS_RETCODE ASC_loadCommParams(CS_CHAR devType, const CS_CHAR *clli, const CS_CHAR
*device, COMM_PARAM_ST **commParams))
```

Arguments:

- **devType:** The type of device to load the parameters.

- **clli:** The host clli to load the parameters.
- **device:** The device ID to load the parameters.
- **commParams:** This is a pointer to the created list of communications parameters. Upon failure, or if there are no communications parameters, commParams is set to 0.

Return Values:

- **CS_SUCCEED:** Operation was successful.
- **CS_FAIL:** Operation failed.

CMD_comm_init

This function initializes the communications interface library. When interface-specific State Table actions are required by the NEP, you use this function to register the actions with CMD_user_actions.

This function is only required if you are using the ASC communication interface. In other words, if you are building your own communication interface, this function is not required.

For more information on CMD_user_actions, refer to "CMD_user_actions."

Syntax:

```
CS_RETCODE CMD_comm_init(void)
```

Return Values:

- **CS_SUCCEED:** Initialization of the NEP was successful.
- **CS_FAIL:** Initialization failed.

CMD_connect_port

This function opens the connection to the device specified by the port information structure. It registers an association between the command processor initiating the connect and the device. This function is supplied by the NEP application. However, if the NEP application links to libasccomm, the custom NEP does not need to define this function.

Syntax:

```
CS_RETCODE CMD_connect_port(CMD_PORT_INFO *port)
```

Arguments:

- **port:** Pointer to the port information data structure.

Return Values:

- **CS_SUCCEED:** Connection was successful.
- **CS_FAIL:** Connection attempt failed.

CMD_disconnect_port

This function closes the connection to the device specified by the port information structure. This function is supplied by the NEP application. However, if the NEP application links to libasccomm, the custom NEP does not need to define this function.

Syntax:

```
CS_RETCODE CMD_disconnect_port(CMD_PORT_INFO *port, SRV_OBJID cmd_qid)
```

Arguments:

- **port:** Pointer to the port information data structure.
- **qid:** Auxiliary command processor message queue ID.

NEP configuration

The interface to the static configuration database tables uses a function-based interface instead of SQL insert scripts. The function-based interface reduces the dependency between administrators who configure the system and product developers who need to make changes to the static tables to support new functionality.

This section lists the descriptions, parameters, and results for the NEP configuration actions. It includes the delete, list, and new procedures interface definitions.

NEP_add_feat

This function maps generic feature names to the feature name for the switch in tbl_march_feat.

Table 5-1 NEP_add_feat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
generic_feat	The generic feature name.	Yes	I
tech	The technology type of the NE or SRP with which the Interpreter is to interact.	Yes	I
switch_feat	The switch feature name.	Yes	I

NEP_add_parm

This function adds a provisioning parameter to the NEP database (tbl_march_rpm).

Table 5-2 NEP_add_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
type	The type of message formatting. Possible values include: • H: host parameters • HN: host/nxx parameters • HU: host/usoc parameters • HUN:host/usoc/nxx parameters • HF: host/feature parameters • HP: host	Yes	I
mcli	The host network element.	Yes	I
nxx	The Central Office code.	Yes	I
usoc	The Universal Service Order Code.	No	I
feat	The switch feature name (this is not the generic feature name).	No	I
lcc	The line class code.	No	I
pname	The parameter name. When the type is HP, RCCF, or NACT then this parameter is the USO PIC, remote activation CCF, or no activation CCF, respectively.	Yes	I

NEP_del_feat

This function deletes a mapping from a generic feature name to the feature name for a switch type from tbl_march_feat.

If you invoke the procedure without any parameters, all rows in the database table are deleted.

Table 5-3 NEP_del_feat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
generic_feat	The generic feature name.	No	I
tech	The technology type of the NE or SRP with which the Interpreter is to interact.	No	I
switch_feat	The switch feature name.	No	I

NEP_del_parm

This function deletes provisioning parameter(s) from tbl_march_rpm.

If you invoke the function without any parameters, all the rows in the database table are deleted.

To delete all USOC parameters for all hosts, the following command can be used:

```
NEP_del_parm type = HU.
```

Table 5-4 NEP_del_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
type	The type of message formatting. Possible values: • H: host parameters • HN: host/nxx parameters • HU: host/usoc parameters • HUN:host/usoc/nxx parameters • HF: host/feature parameters • HP: host pic conversion • HUL: host/usoc/lcc parameters • HUNL: host/usoc/nxx/lcc	No	I
mcli	The host network element.	No	I
nxx	The Central Office code.	No	I
usoc	The Universal Service Order Code.	No	I
feat	The switch feature name (this is not the generic feature name).	No	I
lcc	The line class code.	No	I
pname	The parameter name. Note that when type is HP, RCCF, or NACT then this parameter is the USO PIC, remote activation CCF, or no activation CCF, respectively.	No	I

NEP_show_feat

This function displays the mapping of generic feature names to switch feature names based upon switch type. This information is contained in tbl_march_feat.

If you invoke the procedure without any arguments, all mapping records are displayed.

Table 5-5 NEP_show_feat Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1	Oracle Database Ref Cursor.	Yes	I/O
RC2	Oracle Database Ref Cursor.	Yes	I/O
generic_feat	The generic feature name.	No	I
tech	The technology type of the NE or SRP with which the Interpreter is to interact.	No	I

Table 5-6 NEP_show_feat Results

Name	Datatype	Description
generic_feat	TYP_feat	The generic feature name.
tech	TYP_swtype	The technology type of the NE or SRP with which the Interpreter is to interact.

Table 5-6 (Cont.) NEP_show_feat Results

Name	Datatype	Description
switch_feat	TYP_feat	The switch feature name.
generic_feat	TYP_feat	The generic feature name.
tech	TYP_swtype	The technology type of the NE or SRP with which the Interpreter is to interact.
switch_feat	TYP_feat	The switch feature name.

NEP_show_parm

This function is used to show provisioning parameter(s) contained in tbl_march_rpm.

If you invoke the function without any arguments, all the provisioning parameters are displayed. To show all USOC parameters for all hosts, use the following command:

```
NEP_show_parm type = "HU"
```

Table 5-7 NEP_show_parm Parameters

Name	Description	Req'd	(I)nput/ (O)utput
RC1 to RC8	Oracle Database Ref Cursor.	Yes	I/O
type	The type of message formatting. Possible values: • H: host parameters • HN: host/nxx parameters • HU: host/usoc parameters • HUN:host/usoc/nxx parameters • HF: host/feature parameters • HP: host pic conversion • HUL: host/usoc/lcc parameters • HUNL: host/usoc/nxx/lcc	No	I
mcli	The host network element.	No	I
nxx	The Central Office code.	No	I
usoc	The Universal Service Order Code.	No	I
feat	The switch feature name (this is not the generic feature name).	No	I
lcc	The line class code.	No	I
pname	The parameter name. Note that when type is HP, RCCF, or NACT then this parameter is the USO PIC, remote activation CCF, or no activation CCF, respectively.	No	I

Table 5-8 NEP_show_parm Results

Name	Datatype	Description
host_clli	TYP_clli	The host CLLI.

Table 5-8 (Cont.) NEP_show_parm Results

Name	Datatype	Description
nxx	TYP_nxx	The Central Office code.
usoc	TYP_usoc	The Universal Service Order Code.
feat	TYP_feat	The switch feature name.
lcc	TYP_lcc	The line class code.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
nxx	TYP_nxx	The Central Office code.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
nxx	TYP_nxx	The Central Office code.
usoc	TYP_usoc	The Universal Service Order Code.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
feat	TYP_feat	The switch feature name.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
usoc	TYP_usoc	The Universal Service Order Code.
lcc	TYP_lcc	The line class code.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.
host_clli	TYP_clli	The host CLLI.
nxx	TYP_nxx	The Central Office code.
usoc	TYP_usoc	The Universal Service Order Code.
lcc	TYP_lcc	The line class code.
param_lbl	TYP_name	The parameter name.
param_vlu	TYP_value	The parameter value.

NEP administration

This section lists NEP administration functions.

RPC screen_dump

This function dumps the virtual screen for the specified port to a file.

Syntax:

```
screen_dump  @device, @filename
```

Table 5-9 RPC screen_dump Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I
@filename	The name of the file to direct the virtual screen content.	Yes	I

RPC screen_enable

This function enables the tbl_resource_pool to dump data from the virtual screen, assigned to each device (port), into the specified file. If the file already exists, the data shall be appended to the file.

This RPC shall be useful during testing and debugging of devices.

> **Note:**
>
> You must ensure that sufficient space is available on the file systems. Depending on the switch technology, the virtual screen outputs can be quite bulky, with potential to overflow the file storage resources quickly. For this reason, this RPC should only be used for testing or debugging, and not for prolonged periods of time.

Syntax:

```
screen_enable  @device, @filename
```

Table 5-10 RPC screen_enable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I
@filename	The name of the file to direct the virtual screen content.	Yes	I

RPC screen_disable

This function disables continuous dumping of the virtual screen diagnostics, enabled by RPC screen_enable, into the file.

Syntax:

```
screen_disable @device
```

Table 5-11 RPC screen_disable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I

RPC line_enable

This function enables a continuous line diagnostics dump of the specified device (port), from the tbl_resource_pool of $SARM_DB.

> **Note:**
>
> You must ensure that sufficient space is available on the file systems. Depending on the switch technology, the virtual screen outputs can be quite bulky, with potential to overflow the file storage resources quickly. For this reason, this RPC should only be used for testing or debugging, and not for prolonged periods of time.

Syntax:

```
line_enable  @device, @filename
```

Table 5-12 RPC line_enable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I
@filename	The name of the file to direct the dump content.	Yes	I

RPC line_disable

This function disables the line diagnostics dump of the specified device (port), from the tbl_resource_pool of $SARM_DB.

Syntax:

```
line_disable @device
```

Table 5-13 RPC line_disable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I

RPC edd_diag

This function allows the External Device Driver (EDD) to dump data of the specified debugging type into the file. Depending on EDD implementation, the requested output can overwrite the file or append to the file.

Syntax:

```
edd_diag @device, @filename, @type
```

Table 5-14 RPC edd_diag Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I
@filename	The name of the file to direct the content of the diagnostics.	Yes	I
@type	Debugging data type. Possible values are: • start_dump • stop_dump • dump_info • start_dump_all • stop_dump_all	Yes	I

RPC enable

This function allows an administrative enabling (setting) of the device (port), from tbl_resource_pool of $SARM_DB.

Syntax:

```
enable @device
```

Table 5-15 RPC enable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I

RPC disable

This function allows an administrative disabling of the device (port), from tbl_resource_pool of $SARM_DB. For the device (port) in the connected state, the disconnection is issued.

Syntax:

```
disable @device
```

Table 5-16 RPC disable Parameters

Name	Description	Req'd	(I)nput/ (O)utput
@device	The name of the device (port) from the table tbl_resource_pool of $SARM_DB	Yes	I

Switch configuration library

The following section provides details on the interface for the Switch Configuration library. There are no global variables or inline functions for this library.

ASC_libnecfg_init

This function initializes the Switch Configuration library. Currently, this function adds support for March, DMS, and blackout-related functions. This routine must be called from the CMD_comm_init() function in the custom NEP application. For more information on CMD_comm_init, see "CMD_comm_init."

Syntax:

```
CS_RETCODE ASC_libnecfg_init(void)
```

Return Values:

- **CS_SUCCEED:** Switch configuration library was successfully initialized.
- **CS_FAIL:** Initialization has failed. The NEP server diagnostics recognize the problem.

Protocol-specific libraries

This section describes the TL1 library used by the NEP.

The ASAP State Tables provide a means for provisioning network elements using Transaction Language 1 (TL1) Interface. ASAP enhances the TL1 support capability by providing action functions to simplify State Table programming. The following information on TLI is included:

- Design Assumptions
- Functional Architecture
- Technical Architecture
- TL1 State Table API
- TL1 State Table Action Examples

> **Note:**
>
> Enhanced Transaction Language 1 (TL1) conforms to Bellcore standard, version TR-NWT-000831, Issue 3.

Design assumptions

The development of new action functions is based on the following assumptions:

- TL1 implemented by the NE vendor conforms with Bellcore TR-NWT-000831.
- Provisioning using TL1 may be established over several different communication protocols, including: modem dial-up line, RS232 serial line, X.25, etc. Besides currently supported dial-up line, RS232 serial line, AIX X.25 interface and TCP/IP Telnet, ASAP will support an X.25 interface on SUN Solaris.
- X.25 PAD connection is not supported.
- ASAP supports command line interface of TL1 on NEs, but not menu or GUI interface.
- Autonomous messages are not supported.

Functional architecture

The TL1 language structure consists of input and output messages that are described by functional parameter blocks and parameters in those blocks. Although current core ASAP action functions support the TL1 language, ASAP provides an additional set of actions to simplify the utilization of TL1 in State Tables.

TL1 support in ASAP is provided in the NEP through TL1 action functions that address the issues described in the following sections.

Building message block

A TL1 action automatically inserts ASDL parameters (more precisely State Table variables) into the corresponding blocks of a TL1 input message, according to the prefix of the parameters. The dot notion is used to specify the prefix. For example, BLK_A.OE denotes that the parameter OE is in the block BLK_A. While ASAP provides default names for a block, these default names can be overridden in the State Table programs.

If there is no parameter for an optional block, an empty block is formed by the action function.

Parameter handling

The TL1 action constructs a parameter block with name-defined parameters. The exception is in the case of a block that allows only position-defined parameters, for instance, Target Identifier Block.

Submit TL1 input message

Once a TL1 input message is constructed, you invoke a TL1 process action to trigger the NE to process the message. The process action blocks the State Table until all output messages are received or a time-out occurs. When the action times out, it aborts the process.

Processing output message

Some TL1 output messages are machine-parsable. TL1 actions can access TL1 output message blocks.

A parser parses both name-defined parameters and position-defined parameters of output messages. Parsed parameters are saved as State Table variables with either parameter names or position identifiers. You can then access output data items directly through State Table variables.

Technical architecture

TL1 support in ASAP is provided in the NEP through a State Table program executed in a thread. The technical implementation of TL1 support consists of a set of action functions and a parser.

Action functions are implemented using current ASAP libraries that provide the basic building blocks for action functions.

TL1 support constructs TL1 input messages using name-defined parameters wherever valid (based on the Bellcore definition). This method is used because of its suitability for machine-to-machine interfaces.

The parser is a new layer built on the top of the ASAP communication unit: multiple protocol manager that is transparent to all communication protocols. This plug and play layer ensures that the NEP is fully backward-compatible.

TL1 actions interface with an NE in an interactive mode. Typically, it sends a command to a network element and waits for responses. In addition to this synchronous processing of each State Table program, ASAP processes more than one State Table program simultaneously through multiple connections to network elements.

TL1 State Table API

ASAP provides a set of actions to help implement TL1 in ASAP State Tables. If the TL1 language provided by the NEs does not fully conform with TR-NWT-000831, this set of “TL1-Convenient" actions cannot be used; generic actions provided by ASAP must be used instead.

In State Table programming, there is a significant difference in the handling of input and output messages from TL1. These differences are described below. This section frequently references TL1 language syntax defined in TR-NWT-000831. For more detailed information, please refer to TR-NWT-000831 itself.

A sample of the characters used in TR-NWT-000831 is shown below to describe the TL1 language syntax.

- <> – Encloses a symbol specifier.
- **[]** – Encloses an optional symbol.

- **()** – Encloses a group of symbols for the following operators.
- ***** – Is a postfix operator meaning that the preceding symbol or group of symbols can occur zero or more times.
- **^** – Indicates a space.
- **+** – Is a postfix operator meaning that the preceding symbol or group of symbols can occur one or more times.
- **|** – Is the infix operator meaning that either the preceding or succeeding symbol can occur, but not both in succession.

Input messages

The TL1 input message has the following format:

```
<command_code>:<staging_parameter_block>:<message_payload_block(s)>;
```

where staging_parameter_blocks contains:

```
:[<target identifier>]:[<access identifier(s)>]:
<correlation tag>:[<general block>]:
```

and message_payload_block(s) contains zero or many data blocks.

Parameters in each block can be parameters that are defined by position or name depending on the rule of blocks or your defined preference. The following provides examples of these two types of parameters in a block.

Position-defined parameters:

```
:123, ON, abc:
```

Name-defined parameters:

```
:P1=123, P2=ON, P3=abc:
```

Block name

To automatically insert parameters into the correct block when constructing a TL1 message, pre-agreed block names are used. ASAP internally keeps default block names, but you can choose block names by overriding the defaults in the State Table programs. To override default block names, pass new block names as arguments to the TL1 action. The following block names can be modified:

Table 5-17 TL1 Block Names

TL1 Blocks	Default Block Base Name
<general block>	TL1_GB
<message payload block(s)>. Since this block can have many occurrences, an array identifier [n] is appended to the base name forming the block name. For example, the base name is TL1_MSG, then you can have TL1_MSG[1], TL1_MSG[2], TL1_MSG[3], etc.	TL1_MSG

For all parameters in <general block> block, the parameters can have names such as TL1_GB. ON, TL1_GB.DATE, etc. To put the parameter LINE into the first <message payload block>, the parameter is TL1_MSG[1].LINE.

If there is no parameter found for a certain block, the action constructs an empty block or returns an error if it is a mandatory block.

The message payload block must be in sequence. The action function stops adding message blocks into a TL1 message whenever there are no remaining parameters for that block. If there are no parameters for <message payload block> two, TL1_MSG[2], the action function will not append TL1_MSG[3] even if there are parameters for it.

Some vendors add blocks (not in conformity with the recommendation), but do not use them, so that empty blocks are required. To do this, it is necessary to create dummy parameters to generate empty message payload blocks.

Parameter name

A variable name must have a block name and a TL1 parameter name. The TL1 action strips off the block name before it constructs a block. The TL1 parameter name is the name that is inserted into the corresponding block.

For example, in the block TL1_MSG[1] there are three parameters:

```
:OC=OC3, GOS=COT_GOS4, PST=OOS:
```

You must have the State Table variables TL1_MSG[1].OC=OC3, TL1_MSG[1].GOS=COT_GOS4, and TL1_MSG[1].PST=OOS.

Output messages

TL1 acknowledgments consists of <acknowledgment code> and <ctag> and the TL1 response consists of a response header, an identification portion, and a text block.

```
<header> <response identification> [<text block>] <terminator>
```

where the following formats apply:

Header

```
<cr><lf><lf>^^^<sid>^<year>-<month>-
<day>^<hour>:<minute>:<second>
```

Response Identifier

```
<cr><lf>M^^<ctag>^<completion code>
```

Text Block

```
((<cr><lf>^^^<unquoted line>)|(<cr><lf>^^^<quoted line>)|(<cr><lf>^^^<comment>))*
```

TL1 actions provide the means to process TL1 output messages. The messages are automatically parsed and stored in State Table variables. ASAP provides default base names for the variables, but they can be overridden in State Table programs. An identifier is used as a sub-name to further identify data items in the block. The following table lists the base names and identifiers.

Table 5-18 TL1 Base Names and Identifiers

Default Base Name	Identifier	Description
TL1_ACK	CODE CTAG	<acknowledgment code> <ctag>
TL1_HDR	SID DATE TIME	<sid> in <header> <year>-<month>-<day> in <header> <hour>:<minute>:<second> in <header>
TL1_RESP_ID	CODE CTAG	<completion code> in <response identification> <ctag> in <response identification>
TL1_QUOTED	TL1 Block B[i] and Parameter Name or P[m]	<quoted line> in <text block> The block inside <quoted line> is identified as the position block, so the first block is B[1], the second is B[2] etc. Since this block can appear many times, [n] is appended to the base name. For example: name-defined: TL1_QUOTED[1].B[1].id position-defined: TL1_QUOTED[1].B[2].P[1]
TL1_UNQUOTED	TL1 Block B[i] and Parameter Name or P[m]	<unquoted line> in <text block> If it is a name-defined parameter, the parameter name is used as the identifier. If it is a position-defined parameter, P[m] is used, where m identifies the position. The block inside <quoted line> is identified as the position block, so the first block is B[1], the second is B[2] and so on. Since this block may appear many times, [n] is appended to the base name.For example: name-defined: TL1_QUOTED[1].B[1].id position-defined: TL1_QUOTED[1].B[2].P[1]
TL1_COMMENT	not applicable	<comment> in <text block> Since this block may appear many times, [n] is appended to the base name.

All variables that save parsed data items have the format base_name.identifier. The exception is TL1_COMMENT.

For example, TL1_ACK.CODE=IP. TL1_QUOTED, TL1_UNQUOTED, and TL1_COMMENT can appear zero or more times. Therefore an array identifier [n] is appended to base name forming a block name. For example, TL1_UNQUOTED[2].B[1].P[3]=5 means that 5 is the third parameter of the first block in the second <unquoted line> block.

An array index [i] is appended to the base name, block and position-defined parameter even if there is only one occurrence of the item. For example, if there is only one block in <unquoted line>, B[1] is used.

After a variable is assigned, you can access it in the same way as other State Table variables.

TL1 State Table action functions

TL1 action functions allow you to use fewer actions to implement TL1 in State Tables only. Normally, several general action calls and more sophisticated programming techniques are required, especially when parsing TL1 output messages. The actions provided are:

- TL1_BUILD_MSG
- TL1_PROCESS_MSG
- TL1_BUILD_TSN

TL1_BUILD_MSG

The TL1_BUILD_MSG action is used to construct a TL1 message from State Table variables.

See also TL1_PROCESS_MSG, BUILD_STRUCT.

This action constructs a TL1 message using state table variables.

- **MSG_ID** – Used to save the identifier of the input message
- **%CMD** – TL1 command code
- **%TID** – TL1 target identifier
- **%AID** – Access identifier
- **%CTAG** – TL1 correlation tag
- **%TID** – Optional argument

The following optional arguments are used to override default block base names. If changed, they both must be changed.

The default base names are:

- **TL1_GB** – General block
- **TL1_MSG** – Message payload block(s)

Upon invocation, the **TL1_BUILD_MSG** action function finds the variables matching the prefix block names among all state table variables. The matching parameters are then inserted into the corresponding blocks.

Since message payload block(s) can appear more than once, an array identifier is appended to the base. This identifier forms the block name. For example, TL1_MSG[1] identifies the first base name, TL1_MSG[2] identifies the second, and so on.

The action function searches for matching base name parameters until there is no match for that array number. For example, if there is a parameter TL1_MSG[1].name and TL1_MSG[3].value, only the name parameter is added.

Colons in the action syntax are action delimiters and do not pertain to TL1 block delimiters. The arguments in the last square bracket are the base names used to replace default base names; however, the arguments before it are the data which is inserted into the TL1 input message.

TL1_PROCESS_MSG is used to send this input message to the NE and then retrieve the resulting TL1 output message.

Errors:

State Table program execution ends in failure if you do not provide correct information. You must provide a command and a CTAG. Both the TID and AID are optional, however, the colons between them are not.

Syntax:

```
TL1_BUILD_MSG 'MSG_ID=%CMD:[%TID]:[%AID]:%CTAG [: %GB_BASE:%MSG_BASE]'
```

Example:

The following example displays OC and PST going into the first message payload block. You must first construct parameters:

```
TL1_MSG[1].OC=OC3 and TL1_MSG[1].PST=OOS
```

The following example displays a portion of a State Table program using the action:

```
# Construct a message using saved values for AID, TID and inputted values for
command and CTAG. Default base names for general block and message block will be
used.
...
1000 TL1_BUILD_MSG '%MSG_1=UPDATE:%TID:%AID: C10-10'
# Trigger the NE to process message and wait 60 seconds for output message
before it times out

1100 TL1_PROCESS_MSG '%MSG_1' 60
...
```

TL1_PROCESS_MSG

The **TL1_PROCESS_MSG** action sends the target NE a TL1 input message and then processes the output message.

This action must be used only after a message is constructed with **TL1_BUILD_MSG**, otherwise it may cause a failure of the program.

Once the output message is received, the message is parsed and information is stored in state table variables. The variable names consist of two parts: base name and identifier. The action provides default base names for variables, but you can change base names through variable or constant arguments presented in square brackets. If you are changing base names, you must either change all of them or none of them.

Each type of argument exhibits different behavior. Variable arguments represent State Table parameters. Constant arguments are static parameters that have no external reference.

Variable Argument:

```
2030  CONCAT '%TEST1=LABEL1'
2050  CONCAT '%TEST2=LABEL2'
2080  CONCAT '%TEST3=LABEL3'
2100  CONCAT '%TEST4=LABEL4'
2130  CONCAT '%TEST5=LABEL5'
2140  CONCAT '%TEST6=LABEL6'
3000  TL1_PROCESS_MSG '%MSG1=%TEST1:%TEST2:%TEST3:%TEST4:%TEST5:%TEST6' 60
```

Constant Argument:

```
3000  TL1_PROCESS_MSG '%MSG1=LABEL1:LABEL2:LABEL3:LABEL4:LABEL5:LABEL6' 60
```

Although they demonstrate different approaches, both examples have the same end result: the change of a base name. The base names as shown in both examples change to LABEL1 through LABEL6.

For example, if you want to change the default base names when storing the TL1 output message values, you would set up the State Table as follows:

```
1000 CONCAT '%ACK=AAA'
1100 CONCAT '%HDR=HHH'
1200 CONCAT '%RESP=RRR'
1500 TL1_PROCESS_MSG'%MSG1=%ACK:%HDR:%RESP:QUOTED:UNQUOTED:COMMENT'
```

When storing the output message values, the following values are valid:

- Ack message – AAA.CODE
- Header message – HHH.SID
- Response message – RRR.CODE
- Quoted message – QUOTED.B[1].P[1]
- Unquoted message – UNQUOTED.B[1].P[1]
- Comment message – COMMENT[1]

The timeout value specifies the maximum time of execution for this action function.

After the successful execution of this action, information can be accessed through these defined variables. The following table displays the relationship between an argument and output message.

Table 5-19 Argument - output message relationships

Argument	Output Message
ACK	<acknowledgments>
HEADER	<header>
RESP_ID	<response identifier>
QUOTED	<quoted line>
UNQUOTED	<unquoted line>
COMMENT	<comment>

Errors:

Communication problems and timeouts trigger ASAP to automatically reprocess the related ASDL later. If one base name is changed, all of the other base names must contain a value in the action string.

Failure to execute within the timeout value specified causes a communication failure.

Syntax:

```
TL1_PROCESS_MSG '%MSG_ID[=%ACK:%HEADER:%RESP_ID:%QUOTED:%UNQUOTED:%COMMENT]' timeout
```

Example:

```
# Construct a message
...
1000 TL1_BUILD_MSG '%MID=%UPDATE:::RDBKNJNVK01'
```

```
# Process the message and wait the output message for 120 sec. before time-out
1110 TL1_PROCESS_MSG '%MID' 120
# Check results
1200 IF_THEN '%TL1_RESP_ID.CODE==COMPLD'
1300 TL1_BUILD_MSG '%MSG2 = %RTRV: %TID: %AID: C11'
# this is an example where the default base names are changed. Note that both
constants and State Table variables can be used.
1400 TL1_PROCESS_MSG '%MSG2 = %ACK:HHH:%RESP:'%QUOTED:TL1_QUOTED2:%COMMENT:60
1500 ELSE
...
1600 END_IF
...
```

TL1_BUILD_TSN

Constructs a TL1 test session between an OS and an NE using state table variables. in this way, an AID value is an alias which represents the tested objects. Bellcore TR-NWT-000831 provides some examples of how to use this feature.

- %**MSG_ID** – Used to save the identifier of the input message.
- **CMD** – TL1 command code.
- **TID** – TL1 target identifier.
- **AID** – TL1 access identifier.
- **CTAG** – TL1 correlation tag.
- **TID** – An optional argument.

The last optional parameter is a base name used to automatically construct parameter blocks. All State Table variables matching the block name (base name + [n]) is inserted into the block. The default base name is **TL1_MSG** and can be changed using the optional **MSG_BASE** argument.

TL1_PROCESS_MSG must be used to retrieve the TL1 output message after execution of this action.

Errors:

Program execution ends with failure if you do not provide correct information. If the communication channel is not available, the ASDL that triggers this action is retried.

Syntax:

```
TL1_BUILD_TSN                    '%MSG_ID=%CMD:[%TID]:%AID:%CTAG:%TSN
[:%MSG_BASE]'
```

Example:

```
# Construct a test session using command 'CONN-TACC', no TID, AID '3-24' CTAG
'1234', and test session 'T1'
...
1100 TL1_BUILD_TSN '%MID=CONN-TACC::3-24:1234:T1'
...
```

External device driver

This section provides details of the External Device Driver (EDD), API detailed design, and the design of the sample application using the API.

The External Device Driver (EDD) acts as a gateway to transmit data between the network elements (NEs) and the network element processors NEP. The EDD is used for terminal emulation type and message type communication.

The EDD consists of:

- driver in the NEP – Known as a Generic Driver
- Independent Process – Known as an External Device Driver (EDD)

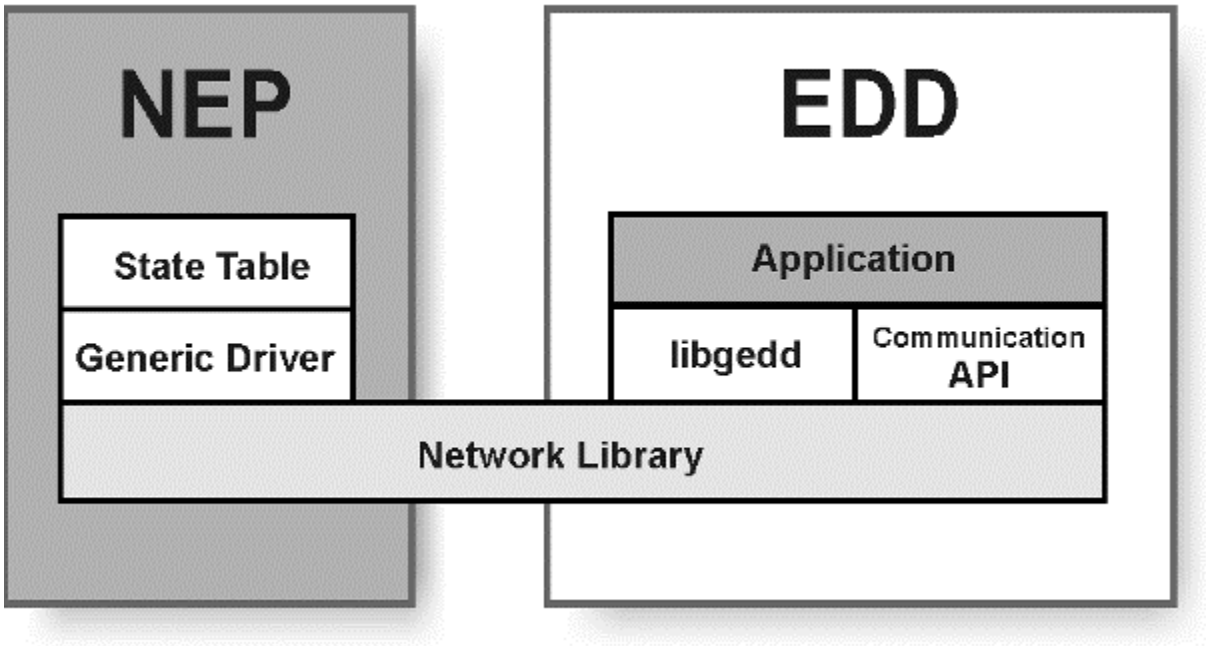
The NEP driver and Independent Process are both referred to as a system of the EDD.

The EDD application uses the non-UNIX device API to manage the communication with network elements. The UNIX socket is used to transfer data between the generic driver and the EDD. The generic driver provides the NEP application a communication vehicle to interface with libgedd.

External device driver architecture

The External Device Driver (EDD) is an independent UNIX process. ASAP requires one EDD for each non-UNIX device communicating with the NEs. The EDD acts as a gateway and forms pseudo connections between the NEP and the NEs. The EDD ensures that the data transmitted from the NEP over a certain connection is forwarded through the associated connection to the NE for that connection. This is a one-to-one relationship between the connections on both sides of the EDD.

Figure 5-1 External Device Driver Architecture



The EDD application interfaces with the NEP using libgedd and with network elements using the appropriate communication API. As a gateway, the EDD is an event-driven device. It is a single-thread process which monitors data from two directions (NEP and NEs) at the same time.

The libgedd uses a signal, poll, or application approach. Each EDD application uses one approach. It is important to select the approach specific to your requirements and only some applications have a choice.

Signal approach

With the signal approach, the EDD application has no control over the events that transmit data from the NEP, however, a UNIX signal interrupts the EDD when data arrives. The EDD application must be written to call a function in libgedd to retrieve the data whenever it detects an interruption by the SIGIO signal. Libgedd maintains an internal data structure to store the connection information. Each connection has its own node of the structure and is identified by an index. The EDD application must use the functions in libgedd to update and access this data structure if it is necessary.

There is a limitation for this approach. Most UNIX functions are written to return the control to the caller when a signal goes off. If the monitor of the communication API is written in this way, the signal approach is used. Some functions do not pass the control back to the caller even if there is a signal, but simply reprocess internally. If so, the only choice is to repeatedly poll on both connections. This creates a problem with performance.

Poll approach

The EDD application must pass the UNIX file descriptor associated with each connection (EDD to NEs) it maintains to the libgedd. The libgedd polls on all connections. The poll function returns an index to the application to identify the connection over which the data is received. The application uses routines provided by the libgedd to retrieve and send data from the NEP. The communication API provides routines to forward and receive data from the NEs.

Application poll approach

In this approach, the application polls the connections (EDD to NEP) on behalf of the libgedd. Once it detects an event on a connection, it uses the routines provided to retrieve data.

Data architecture

The EDD uses the following data types:

- EDD Information Abstract
- Parameter Abstract
- Debugging Abstract
- Generic Driver Abstract

EDD information abstract data type

This structure describes the data used by libgedd to track connections to its managers and the data used by an EDD application to track connections to its managers. The library libgedd owns this structure and provides a set of routines for the application to access it. No application should access this structure directly.

Syntax:

```
typedef struct edd_info_st {
int fd;
int node_idx;
int node_ind;
CS_BOOL in_use;
int debug;
FILE *debug_fptr;
char debug_fname[EDD_DEBUG_FNAME_L+1];
char device[CMD_PROC_DEV_L+1];
CS_BINARY *buf;
int len;
int comm_type;
EDD_COMM_PARAM_ST *comm_param;
int comm_param_len;
void *appl_data;
void (*appl_free)( void *appl_data );
}EDD_INFO_ST;
```

Public members:

- **fd:** File descriptor of the connection.
- **node_idx:** Index identifies the connection.
- **node_ind:** Identifies whether a connection is to the NEP or NE.
- **in_use:** Identifies whether this entry is used.
- **debug:** Identifies whether the debugger is turned on.
- **debug_fptr:** File pointer of the debug file that contains dumped information.
- **debug_fname:** Debug file name.
- **device:** Device name in NEP related to this connection.
- **buf:** Temporary place to save the data coming from NEP.
- **len:** Length of data saved in buf.
- **comm_param:** Communication parameters for this connection. These parameters are saved in the ASAP database and sent to the EDD by the NEP.
- **comm_param_len:** Number of parameters in comm_param.
- **appl_data:** Pointer to data saved by the EDD application.
- **appl_free:** Pointer to a function of the EDD application. When the connection is removed, the libgedd calls it to remove appl_data.

Parameter abstract data type

This structure describes communication parameters. It is used by both the NEP and EDD.

Syntax:

```
typedef struct {
char label[EDD_PARAM_LABLE_L+1];
char value[EDD_PARAM_VALUE_L+1];
}EDD_COMM_PARAM_ST
```

Public members:

- **label:** Label of the parameter.

- **value:** Value of the parameter.

Debugging abstract data type

This structure describes data used for the debugging process.

Syntax:

```
typedef struct {
char device[CMD_PROC_DEV_L+1];
char file_name[EDD_DEBUG_FNAME_L+1];
int type;
}EDD_DEBUG_MSG;
```

Public members:

- **device:** Device name that identifies the connection.
- **file_name:** File name that information is dumped into.
- **type:** Identifies the type of data to dump.

Generic driver abstract data type

This structure describes information used by the generic driver in the NEP to manage the connections to the EDD.

Syntax:

```
typedef struct {
CS_CHAR socket_client;
CS_USHORT sa_family;
CS_CHAR host_name[EDD_HOST_NAME_L+1];
CS_CHAR host_ipaddr[EDD_HOST_IPADDR_L+1];
CS_USHORT port;
CS_CHAR *comm_params;
CS_INT len;
CS_INT write_timeout;
CS_BINARY *buffered_data;
CS_INT buf_len;
}GENERIC_SESS_DATA;
```

Public members:

- **socket_client:** Identifies whether the process is a server or a client from the socket.
- **sa_family:** Identifies the kind of socket.
- **host_name:** Remote host name the socket is connecting to.
- **host_ipaddr:** Remote host IP address.
- **port:** Remote port number.
- **comm_params:** Pointer to a buffer where the communication parameters are saved.
- **len:** Length of the communication parameter buffer.
- **write_timeout:** Time available for writing to the socket.
- **buffered_data:** Buffering data to be sent while the socket is not available.

- **buf_len:** Length of buffered_data.

Transactions

All data transferred between the NEP and EDD has header and data information. The header identifies the length and type of data to be transferred. The data contains raw data that an application intends to transfer. The data header is hidden from the application. The API interprets the message and passes the necessary data type and data to the application.

Data format

Table 5-20 EDD Data Format

Field	Type	Length (in bytes)	Description
Length	Header	4	Total length of data field.
Data type	Header	4	Identify the type of data.
Raw Data	Data	vary	Data (binary or ASCII). Its length is identified by Length.

The application uses the data types listed below to determine the state of a transaction and communication.

Data type

Table 5-21 EDD Data Types

Message Type	Description
EDD_CONNECT	The NEP requires a new connection to an NE. The connection and option parameters are stored in the data field. When the application receives this message, it establishes a connection with an NE. It then puts monitors the connection.
EDD_ACCEPT	A new connection has been established between the NEP and EDD. This data type is only for Application Poll Approach. When the application receives this message, it retrieves a file descriptor which identifies the newly established connection. The application must monitor this connection on behalf of the API.
EDD_CONNECTED	The EDD sends the NEP the confirmation. There is no data field (length = 0). After a connection to an NE is established, the application sends this message to the NEP to confirm. The application does not directly deal with the message. It calls an API function that constructs the message and sends it.
EDD_DISCONNECT	The NEP requires a disconnect from an NE. There is no data field (length = 0). When the application receives this message, it disconnects from the NE and sends the confirmation to the NEP.

Table 5-21 (Cont.) EDD Data Types

Message Type	Description
EDD_DISCONNECTED	The message is sent to the NEP to: Confirm that it has disconnected from the NE on receiving the disconnecting request from the NEP. Inform the NEP that it has lost the connection to the NE. The application does not directly deal with the message. It calls an API function that constructs the message and sends it.
EDD_DATA	Data sent from the EDD or NEP. When the application receives this message, it uses the API to retrieve and process the data. It passes the data to the NE.
EDD_BREAK_KEY	The NEP requests the application to send a break to the NE. When the application receives this message, it sends a break to the NE.
EDD_OPTION	The NEP requests the application to send a option to the NE.
EDD_NO_ACTION	The application ignores this message, since the API handles the NEP request internally. The application does not see this one for Poll Approach since the API detects the event and handles it without knowledge of the application.
EDD_DEBUG	The NEP requires the EDD to start dumping debug information. This is an internal message which is handled by libgedd to dump some diagnostic data.
EDD_DEBUG_DISCONN	The NEP requires to close a debugging connection.

The EDD transaction involves the following processes:

- Connection process
- Disconnection process
- Forward data from the NEP to the NE
- Forward data from the NE to the NEP.

Connection process

- Once the EDD is started, it opens the socket listening channel and waits for the connection request.
- The generic driver in the NEP sends the socket connection request to the EDD. Once the request is accepted, the generic driver sends the connection request for establishing a connection to the NE. This request has a data type EDD_CONNECT. The communication parameters saved in tbl_comm_param for this connection are transferred to the EDD at the same time.
- On receiving the NE connection request, the EDD saves the communication parameters in the format described by the structure EDD_COMM_PARAM_ST and passes the request to the EDD application.
- The EDD application establishes the connection to an NE identified by the communication parameters. An ACK (EDD_CONNECTED) or NACK (EDD_DISCONNECTED) is sent back to the generic driver by calling a routine in libgedd.

- If the connection is established successfully, it is put under the monitoring provided by one of the approaches.
- If the attempt of establishing a connection fails, a NACK is sent to the NEP and the socket connection to the NEP is closed.

Disconnection process

- If the NEP initiates the disconnection process, the generic driver sends EDD_DISCONNECT to the EDD.
- On receiving the request, the EDD application is informed to close the connection to the NE. The application then sends EDD_DISCONNECTED to the NEP and closes the socket connection to the NEP.
- If the EDD initiates the disconnection process, usually due to a lost connection to the NE, it sends EDD_DISCONNECTED to the NEP and then closes the connection to the NEP.

Forward data from NEP to NE

- Connections to the NEP are being monitored constantly. Whenever an arrival of data from the NEP is detected, the EDD application is informed.
- The EDD application uses a routine in libgedd to retrieve the data.
- The EDD application forwards the data retrieved to a designated NE using the communication API.

Forward data from NE to NEP

- Connections to the NE are being monitored constantly. Whenever an arrival of data from an NE is detected, the EDD application is informed.
- The EDD application retrieves the data using the communication API.
- The EDD application calls a routine in libgedd to forward the data to the NEP.

Functions of libgedd

The library, libgedd, handles the communication between the NEP and the EDD. The library provides a set of functions for the EDD application (EDD AP) to:

- access the internal data structure of the libgedd library
- monitor connections
- transmit data between the NEP and EDD.

The following sections provide details on the functions specific to the signal approach, poll approach, and application poll approach.

For backward compatibility, the function names in the bracket represent the previously used names, and are still valid.

Signal approach

Table 5-22 Signal Approach

Event	Functions	Required	Description
SIGIO signal interrupt.	gedd_signal_get_req (gedd_process_nep_req)	Yes	Retrieve data and data type from the NEP. Possible types are the header types described in "Transactions."
After and before the application monitors connections.	gedd_block_sigio gedd_unblock_sigio	Yes	The EDD application must block the SIGIO signal before it starts to process data and unblock it to monitor the connections.
N/A	gedd_sigio_occurred	Yes	Check if the signal has gone off. This is used to protect the application from missing signals because of race conditions caused by the gap after the application unblocks the signal and monitors connections.
N/A	gedd_sigio_reset	Yes	Reset flag which is set when the signal went off. Used together with gedd_sigio_occurred.

Poll approach

Table 5-23 Poll Approach

Event	Functions	Required	Description
Wait for events	gedd_poll	Yes	gedd_poll monitors all connections and reports the events to the application. It returns: POLL_NE_CONNECT – There is data from the NE. POLL_NEP_CONNECT – There is data from the NEP. POLL_FAIL – This is the system problem. UNIX system calls return error. POLL_TIMEOUT – Before an event occurs, gedd_poll times out. POLL_HANGUP – It detects that the connection is hung up.
After connection to an NE.	gedd_add_fd	Yes	Pass file descriptor to the libgedd to be monitored.

Application poll approach

Table 5-24 Application Poll Approach

Event	Functions	Required	Description
Data comes from the NEP	gedd_appl_poll_get_req (gedd_fetch_nep_req)	Yes	The application retrieves data from the NEP. Possible types returned are those data types described in "Transactions."
After initialization	gedd_get_listen_fd (get_listen_fd)	Yes	The application uses it to obtain the file descriptor that the API uses to listen for the connecting request. The application must monitor this connection for the API.
After receive EDD_ACCEPT	gedd_get_fd	Yes	The application calls this function to extract the file descriptor from the data buffer. After the application gets the file descriptor, it must monitor this connection for the API.

Common functions

Table 5-25 Common Functions

Event	Functions	Required	Description
EDD application startup.	gedd_init	Yes	Initializes libgedd.
Connect to the NE succeed.	gedd_api_connect_ack	Yes	The application confirms to the NEP that connecting to the NE has been completed successfully.
Disconnect from the NE.	gedd_api_disconnect_ack	Yes	The application confirms to the NEP that the disconnection has been successfully completed.
Data comes from NEs and must be forwarded to the NEP.	gedd_send_to_nep	Yes	The application uses this routine to send data to the NEP.
N/A	gedd_set_appl_data	No	Saves connection-related information by an EDD application. The application retrieves this information using gedd_get_appl_data.
N/A	gedd_get_appl_data	No	Retrieves data saved by gedd_set_appl_data. It is strongly recommended to use these functions to save connection-related data.

Table 5-25 (Cont.) Common Functions

Event	Functions	Required	Description
Connection Request	gedd_get_conn_param	Yes	Retrieve communication parameters sent by the NEP. These parameters tell the EDD application which NE to connect and information required to establish the connection. Note: These parameters are stored in the ASAP database table tbl_comm_param and transferred to the EDD from the NEP.

Library functions

This section lists the library functions.

gedd_add_fd

This function allows the application to instruct the API to monitor a connection. An added connection is always a connection to the NE that has a corresponding connection to the NEP.

Syntax:

```
CS_VOID gedd_add_fd (CS_INT conn_idx, int fd)
```

Arguments:

- **conn_idx:** Identifies the connection to the NEP that the new file descriptor is related to.
- **fd:** The file descriptor to be added.

Return Values:

```
gedd_poll
```

gedd_api_connect_ack

See also gedd_api_disconnect_ack.

This function is called after the application successfully connects to an NE. If the NEP does not receive the confirmation in a certain period of time, the connection attempt fails. Since a connection to an NE is always related to a connection to the NEP, the identifier of the connection to the NEP is used as a reference.

Syntax:

```
CS_VOID gedd_api_connect_ack (CS_INT conn_idx)
```

Arguments:

- **conn_idx:** Identifies the connection to the NEP.

gedd_api_disconnect_ack

See also gedd_api_connect_ack.

This function is used in the following scenarios:

- The application calls this function to inform the NEP that establishing a connection with an NE has failed. The NEP tries to reestablish the connection and reprocess the incomplete task
- The application calls this function to confirm to an NEP that a disconnect request has completed (successful or failed).
- The application informs the NEP that it has just lost a connection to the NE. The NEP tries to reestablish the connection and reprocess the incomplete task

Syntax:

```
CS_VOID gedd_api_disconnect_ack(CS_INT conn_idx)
```

Arguments:

- **conn_idx:** Identifies the connection to NEP.

gedd_appl_poll_get_req

See also gedd_get_fd.

This function is only used for the Application Poll Approach and allows the application to retrieve data from the NEP. When the application detects event(s) from connections to the NEP, it must call this function to retrieve data. Based on the data type returned by this function, the application takes proper action.

The following lists the valid data types.

- **edd_connect:** NEP requests a connection to an NE.
- **edd_accept:** A new connection to the NEP is established. The application should get the related file descriptor.
- **edd_disconnect:** NEP requests to disconnect from an NE.
- **edd_data:** Data from the NEP is ready.
- **edd_break_key:** NEP requests to send a break to an NE.
- **edd_option:** NEP requests to send a option to an NE.
- **edd_no_action:** API internally accepts a connection. The application should ignore this event.

Syntax:

```
CS_VOID gedd_appl_poll_get_req(int fd, CS_INT *request, CS_BINARY *buf, CS_INT *len)
```

Arguments:

- **fd:** File descriptor that has data.
- **request:** Returns the data type mentioned in the previous table.
- **buf:** Returns the data from the NEP.

- **len:** Returns the length of the data.

gedd_block_sigio

See also gedd_unblock_sigio, gedd_signal_get_req, gedd_sigio_occurred, and gedd_sigio_reset.

This function is only used for Signal Approach and blocks the signal SIGIO. The application uses this function to prevent signal interruption when it is processing data. It unblocks the signal when it starts to poll connections.

Syntax:

```
CS_VOID gedd_block_sigio(CS_VOID)
```

gedd_get_appl_data

This function allows the application to retrieve the data it saved before using gedd_set_appl_data. This function and gedd_set_appl_data are used together for the application to save the data that is related to a connection.

See also "gedd_set_appl_data."

Syntax:

```
CS_VOID *gedd_get_appl_data (CS_INT conn_idx)
```

Arguments:

- **conn_idx:** Identifies the connection to the NEP.

Return Values:

The data retrieved.

gedd_get_conn_param

This function allows the application to access parameters defined in tbl_comm_param. The application uses these parameters to establish a connection to an NE. This function only returns parameters that relate to the specified connection. Since this function returns a pointer to the parameter, the application does not delete it.

Syntax:

```
CS_CHAR *gedd_get_conn_param( CS_INT conn_idx, CS_CHAR *label)
```

Arguments:

- **conn_idx:** Identifies the connection to the NEP.
- **label:** Parameter label interested.

Return Values:

The value of the parameter.

gedd_get_fd

This function allows the application to retrieve the file descriptor that identifies a connection to the NEP. This function is used only for the Application Poll Approach.

This function must be used after a call to gedd_appl_poll_get_req returns data type EDD_ACCEPT. The application puts the file descriptor into its poll list.

See also "gedd_appl_poll_get_req."

Syntax:

```
int gedd_get_fd(CS_CHAR *buf, CS_INT len) q
```

Arguments:

- **buf:** The buffer returned by a call to gedd_appl_poll_get_req.
- **len:** The length of the data in the buf.

Return Values:

File descriptor.

gedd_get_listen_fd

This function is only used for the Application Poll Approach and allows the application to obtain the EDD listening-file-descriptor. The application must call this function after it calls gedd_init. It must then monitor it on behalf of the API.

See also gedd_init.

Syntax:

```
int gedd_get_listen_fd (CS_VOID)
```

Return Values:

File descriptor.

gedd_poll

This function is only used for the Poll Approach and allows the application to poll all connections to the NEP and NEs. The application calls this function whenever it finishes processing data and this function returns the next event. If it detects events, the application calls gedd_poll_get_req to retrieve data.

Syntax:

```
CS_INT gedd_poll(CS_INT timeout, CS_INT *conn_idx)
```

Arguments:

- **timeout:** Time in milliseconds that gedd_poll should wait for events. If a machine does not support milliseconds, it is rounded up to the nearest legal value available on the system. Possible values are:
  - **0** – Returns immediately
  - **-1** – Waits until there is at least one event
- **conn_idx:** Connection identifier returned by gedd_poll. It identifies which connection receives events. This identifier is used in subsequent API calls.

Return Values:

- **POLL_NE_CONNECT:** Events from connection to NEs.

- **POLL_NEP_CONNECT:** Events from connection to the NEP.
- **POLL_FAIL:** System error.
- **POLL_TIMEOUT:** Specified time is expired while gedd_poll has not detected any event.
- **POLL_HANGUP:** One of connections hangs up.

gedd_poll_get_req

This function is only used for the Poll Approach and allows the application to retrieve data from the NEP. The application must use it to retrieve data when a call to gedd_poll returns events. Based on the data type returned by this function, the application takes the appropriate action.

See also gedd_poll.

The valid data types are:

- **edd_connect:** NEP requests a connection to an NE.
- **edd_disconnect:** NEP requests to disconnect from an NE.
- **edd_data:** Data from the NEP is ready.
- **edd_break_key:** NEP requests to send a break to an NE.
- **edd_option:** NEP requests to send a option to an NE.

Syntax:

```
CS_VOID gedd_poll_get_req(int fd, CS_INT *request,
CS_BINARY *buf, CS_INT *len)
```

Arguments:

- **fd:** File descriptor that has data.
- **request:** Returns the data type mentioned in the previous table.
- **buf:** Returns the data from NEP.
- **len:** Returns the length of the data.

gedd_send_to_nep

This function allows the application to send data to the NEP. If this function, the application disconnects the connection.

Syntax:

```
CS_INT gedd_send_to_nep (CS_INT conn_idx, CS_VOID *buf, CS_INT len)
```

Arguments:

- **conn_idx:** Connection identifier.
- **buf:** Pointer to the data to be sent. The data can be either ASCII or binary as long as the NEP can handle it.
- **len:** The length of the data.

Return Values:

- **cs_succeed:** The data was successfully sent to the NEP.
- **cs_fail:** Either data cannot be delivered or allowed time has expired. It is set by IO_TIMEOUT in ASAP.cfg or default value of three minutes.

gedd_set_appl_data

This function allows the application to save data related to a connection. When the application uses the function, it provides a free function that is used by the API to free up the memory at disconnection time.

See also "gedd_get_appl_data."

Syntax:

```
CS_VOID gedd_set_appl_data( CS_INT conn_idx,
CS_VOID *appl_data, CS_VOID (*free_ap_data)(CS_VOID *appl_data))
```

Arguments:

- **conn_idx:** Identifies the connection.
- **appl_data:** The data the application wants to save.
- **free_ap_data:** The function pointer to the cleanup function. It is called when the connection is dropped.

gedd_sigio_occurred

This function allows the application to check whether a signal is delivered during a period of time.

See also gedd_sigio_reset, gedd_block_sigio, and gedd_unblock_sigio.

Syntax:

```
CS_INT gedd_sigio_occurred(CS_VOID)
```

Return Values:

- **TRUE:** There is a signal.
- **FALSE:** There is no signal.

gedd_sigio_reset

This function resets the flag that is used by gedd_sigio_reset to determine if a signal has gone off.

See also "gedd_sigio_occurred," "gedd_block_sigio," and "gedd_unblock_sigio."

Syntax:

```
CS_VOID gedd_sigio_reset(CS_VOID)
```

gedd_signal_get_req

This function is only used for the Signal Approach and allows the application to retrieve data from the NEP. Once the application has detected an interruption of SIGIO, it calls this function to retrieve data. Based on the data type returned, the application takes proper action.

See also gedd_block_sigio, gedd_unblock_sigio, gedd_sigio_occurred, and gedd_sigio_reset.

The valid data types are:

- **edd_connect:** NEP requests a connection to an NE.
- **edd_disconnect:** NEP requests to disconnect from an NE.
- **edd_data:** Data from NEP is ready.
- **edd_break_key:** NEP requests to send a break to an NE.
- **edd_option:** NEP requests to send a option to an NE.
- **edd_no_option:** API internally accepts a connection. The application should ignore this event.

Syntax:

```
CS_VOID gedd_signal_get_req (int fd, CS_INT *request,
CS_BINARY *buf, CS_INT *len)
```

Arguments:

- **fd:** File descriptor that has data.
- **request:** Returns the data type mentioned in the table.
- **buf:** Returns the data from the NEP.
- **len:** Returns the length of the data.

Return Values:

None

gedd_unblock_sigio

This function is only used for the Signal Approach and unblocks the signal SIGIO. Before the application starts to poll its connections to the NEs, it calls this function to unblock the signal SIGIO. When the poll function is interrupted by SIGIO, the application calls gedd_signal_get_req() to retrieve data.

See also "gedd_block_sigio," "gedd_signal_get_req," "gedd_sigio_occurred," and "gedd_sigio_reset."

Syntax:

```
CS_VOID gedd_unblock_sigio(CS_VOID)
```

Arguments:

None

Return Values:

None

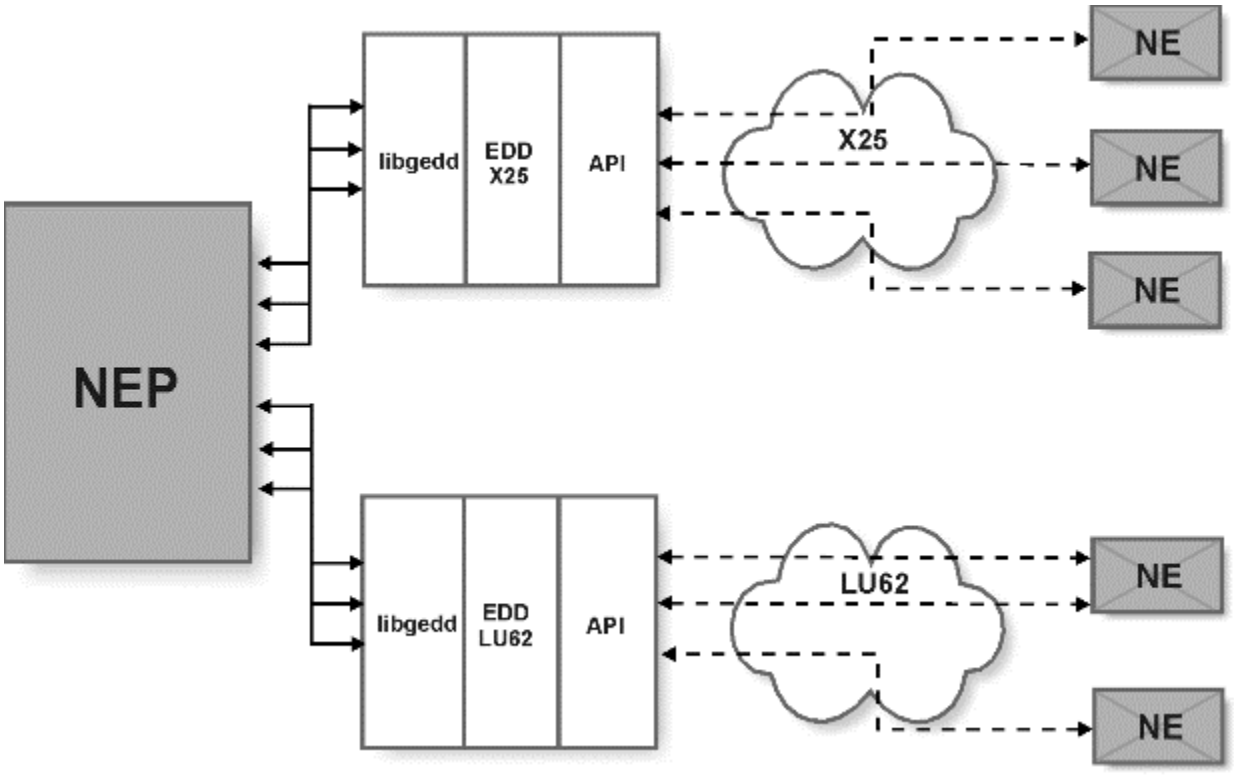
Building an EDD application

An EDD application is built on top of libgedd and network communication APIs. It uses functions provided by libgedd to interface with the NEP and network APIs to interface with NEs.

To generate an EDD application, you enable it to monitor all connections to the NEP and NEs.

Figure 5-2 shows the EDD applications, EDDX25 and EDDLU62, interfacing with NEs through different networks.

Figure 5-2 EDD applications interfacing with NEs



The following scenarios describe how to determine which approach you use to transmit data between the NEP and the EDD.

Using the poll approach

If the network API can convert a connection to the UNIX file descriptor, use the Poll Approach. For the Poll Approach, the application passes the file descriptor to libgedd after it has established a connection to an NE. The application call gedd_poll() monitors all connections.

Using the application poll approach

If the network API can monitor the connections to the NEP on behalf of libgedd, you use the Application Poll Approach. For the Application Poll Approach, the application must monitor all connections including connections to the NEP. For this approach, the application uses APIs to retrieve data to and from the NEP.

Using the signal poll approach

For the Signal Approach, the application uses routines provided by the network API to monitor the connection to the NEs. When data is coming from the NEP, a UNIX signal, SIGIO, interrupts the monitoring. Once the monitoring is interrupted, the application calls routines provided by libgedd to retrieve the data.

Approach examples

The following examples outline how to implement the Signal, Poll and Application Poll Approach in the application. The examples highlight each approach, although you can use alternative ways.

Signal Approach:

```
/* application initialization */
void edd_ap_init(void)
{
   ...
   gedd_init( EDD_SIGNAL_TYPE ); /* signal approach */
   ...
}

void edd_ap_main(void)
{
   ...
   edd_ap_init();
   ...
   while( TRUE ){
gedd_unblock_sigio();
/* Any signal during the block period */
if( gedd_sigio_occurred() ){

gedd_sigio_reset();
cc = EDD_NEP_DATA_IN;

}
else{

alarm( 300 ); /* in case we missed a signal */
/* start to monitor connection to NE */
cc = edd_ap_wait();

}
/* we don't want to be interrupted */
gedd_block_sigio();
alarm(0);
switch( cc ){
case EDD_NEP_DATA_IN:
```

```
edd_ap_int_handler();
break;
case EDD_NE_DATA_IN:
edd_ap_receive();
break;
...
}

   }
   ...
}
void edd_ap_int_handler(void)
{
   ...
   /* get data from all connections they have data to be retrieved */
   while( gedd_signel_get_req( &request, &conn_idx, &buf, &len ) ){
switch( request ){
case EDD_CONNECT:

if( ( conn_id = edd_ap_connection( conn_idx ) )== FAIL ){
gedd_api_disconnect_ack( conn_idx );
else{

/* save conn_id & conn_idx together in "appl_data" */

edd_ap_save_conn_idx( conn_id, conn_idx );
gedd_api_connect_ack( conn_idx );
}
break;
case EDD_DISCONNECT:
/* disconnect from NE and inform the disconnection */
appl_data = gedd_get_appl_data( conn_idx );
edd_ap_disconnection( appl_data->conn_id );
gedd_api_disconnect_ack( conn_idx );
break;


case EDD_DATA:

/* send data to NE */
appl_data = gedd_get_appl_data( conn_idx );
edd_ap_send( appl_data->conn_id, buf, len );
break;

}
...
   }
   ...
}

void edd_ap_receive(void)
{
   ...
   /* this routine finds conn_idx first, then related conn_id */
   conn_idx = edd_ap_next_active_conn();
   appl_data = gedd_get_appl_data( conn_idx );
   edd_ap_retrieve_data( appl_data->conn_id, &buf, *len );
   /* send data to NEP */
   gedd_send_to_nep( conn_idx, buf, len );
   ...
}
```

edd_ap_wait() must detect whether it is awakened by a signal or data coming over the connection from an NE. If it is a signal, **gedd_process_nep_req** must be called to handle the event. This function is also required to block the process even if there is no connection.

The EDD application must associate each connection it manages with the conn_idx that it receives when it is connecting. In the example, this is done using the following routines: **edd_ap_save_conn_idx()** and **edd_ap_next_active_conn()**.

The alarm() function used in the example is necessary, because the SIGIO signal can go off at any time in between the statements, "else{" and "cc = edd_ap_wait". If the signal goes off in this period, the system can go into a dead lock.

In **edd_ap_save_conn_idx**, the **gedd_set_appl_data** function is used to save connection-related data.

Poll approach:

```
void edd_ap_init(void)
{
   ....
   gedd_init( EDD_POLL_TYPE ); /* poll approach */
   ...
}
void edd_ap_main(void)
{
   ...
   edd_ap_init();
   ...
   while( TRUE ){
/* monitor all connections */
cc = gedd_poll( POLL_BLOCK, &conn_idx );
switch( cc ){
/* where data comes from */

case POLL_NEP_CONNECT:
edd_ap_int_handler( conn_idx );
break;
case POLL_NE_CONNECT:
edd_ap_receive( conn_idx );
break;
...
}

   }
   ...
}

void edd_ap_int_handler( int conn_idx )
{
   ...
/* retrieve data from NEP */
      gedd_poll_get_req( conn_idx, &request, &buf, &len ) ){
      switch( request ){
      case EDD_CONNECT:
if( fd = edd_ap_connection( conn_idx, &conn_id ) == FAIL ){
gedd_disconnect_ack( conn_idx );
}
      else {
      /* passes file descriptor to libgedd */
```

```
gedd_add_fd( conn_idx, fd );
edd_ap_set_ap_data( conn_idx, conn_id );
gedd_connect_ack( conn_idx );
}
break;
case EDD_DISCONNECT:
appl_data = gedd_get_appl_data( conn_idx );
edd_ap_disconnection( appl_data->conn_id );
gedd_disconnect_ack( conn_idx );
break;
case EDD_DATA:
appl_data = gedd_get_appl_data( conn_idx );
edd_ap_send( appl_data->conn_id, buf, len );
break;

   }
   ...
}

void edd_ap_receive( int conn_idx )
{
   ...
   ap_data_st = ( AP_DATA_ST *)gedd_get_appl_data( conn_idx );
   edd_ap_retrieve_data( ap_data_st->conn_id, &buf, *len );
/* send data to NEP */
   gedd_send_to_nep( conn_idx, buf, len );
   ...
}
```

The key difference between the routines above and the routines in the Signal Approach section is that the call to gedd_add_fd(conn_idx, fd) right after the EDD application establishes the connection.

The conn_id, in the example, is a connection identifier used by a non-UNIX device API to access the connection to the NEs.

In this example, the programs calls **edd_ap_set_ap_data()** (it calls **gedd_set_appl_data()**) to save the API conn_id and other information into the array of **EDD_INFO_ST**. The **gedd_get_appl_data()** retrieves the conn_id back. In this approach, the application does not build a relationship between its connections and **conn_idx**, since it uses the **conn_idx** as an index to save and retrieve the information it needs. The libgedd takes care of the correlation among the connections.

Application poll approach:

```
void edd_ap_init(void)
{
   /* code here to do the application initialization */
....
   gedd_init( EDD_APPL_POLL_TYPE ); /* poll approach */

 /* pass fd to the application for monitoring */

   edd_add_fd(gedd_get_listen_fd());
   ...
}
void edd_ap_main(void)
{
   ...
   edd_ap_init();
   ...
```

```
   while( TRUE ){
/* monitor all connections */
cc = edd_ap_monitor(&fd);
switch( cc ){
/* check where data comes from and handle it */
case NEP_CONNECTION:

edd_ap_nep_data_handler( fd );
break;
case NE_CONNECTION:
edd_ap_ne_data_handler( fd );
break;
...
}

   }
   ...
}

void edd_ap_nep_data_handler( int fd )
{
   ...
/* retrieve data from NEP */
   gedd_appl_poll_nep_req( fd, &request, &buf, &len ) ){
   switch( request ){
   case EDD_ACCEPT:

new_fd = gedd_get_fd(buf, len);
/* code here to put fd in application poll list */
...
break;


case EDD_CONNECT:
   /* Connect to NE. Assume edd_ap_connection does all works such as puting new
fd under monitoring and connecting to NE */

if ( edd_ap_connection( fd ) == FAIL ){

gedd_disconnect_ack( fd );

else {
gedd_connect_ack( fd );
}
break;

case EDD_DISCONNECT:
appl_data = gedd_get_appl_data( fd );
edd_ap_disconnection( appl_data );
gedd_disconnect_ack( fd );
break;

case EDD_DATA:
appl_data = gedd_get_appl_data( fd );
edd_ap_send( appl_data, buf, len );
break;

   }
   ...
}
```

```
void edd_ap_ne_data_handler( int fd )
{
   ...
/* retrieve data from NE */
   edd_ap_retrieve_data( fd, &buf, *len );

   /* get file descriptor for NEP here */
   ...
/* send data to NEP */
   gedd_send_to_nep( nep_fe, buf, len );
   ...
}
```

The key difference between the above routines and the routines in other approaches is to retrieve the file descriptor and put it under the monitoring. The API does not monitor data from the NEP.

For the Application Poll Approach, the API uses the file descriptor as a connection identifier.

In this example, the program uses the **appl_data** to correlate connections. When a connection request is received, it should save the connection identifier to an NE into **appl_data**. The application can use other ways to do this.

Action functions

The State Table Interpreter interprets script programs (similar to Basic) which are maintained in static database tables. The scripts in these tables control the operation of a State engine, also known as State Tables.

State Tables provide a table-driven user-programmable scripting language to customize the ASAP provisioning process. They can be incorporated into any ASAP application server process, control the ASAP dialog with external systems, and provide a flexible mechanism for you to modify ASAP functionality without changing source code.

State Tables are used extensively in the NEP to interface to network elements and upstream systems respectively. They are also employed in the SRP to facilitate customer controlled translation logic in addition to that provided by the static translation tables.

State Table Components

This section explains the various state table components.

State Table environment

The State Table Environment consists of the following components:

- **State Table Compiler** – allows the State Table developer to create State Table programs in normal UNIX files, and then compile them into suitable database insert scripts. If no ASDL to State Table mapping is defined, ASAP uses the State Table name as the default ASDL name.
- **State Table Optimizer** – as the State Tables are loaded from the database by the Interpreter, they can be optimized for better performance before execution. This is a user-configurable process.
- **State Table Cache** – once a State Table program has been compiled, optimized, and executed, the Interpreter maintains it in a State Table Cache. The next request for that

State Table is read from the cache, not the database, yielding better State Table performance.

The Interpreter also provides an administrative RPC to flush the State Table cache to force a cache reload as the system is running. This enables you to change State Table programs without shutting down the specific application server.

- **State Table Debugger** – A State Table debugger application interacts with the Interpreter to debug State Tables. You can test and debug State Tables in the development phase of a project implementation using a State Table debugger.

  The State Table debugger currently supports the following front-end interfaces:

  – UNIX character-based interface.

  – MS Windows interface, which requires a PC networked to ASAP with the appropriate networking software.

  The State Table debugger provides a command line interface that facilitates the testing of the State Table Interpreter in an ASAP application server. The following features are supported:

  – **Breakpoints** – You can set, clear, or list Interpreter breakpoints at specific line numbers in a State Table program. You can set up to 1000 breakpoints.

  – **Watch Variables** – You can set and configure up to 1000 watch variables. These are compound parameters that are displayed when the State Table execution is stopped.

  – **State Table Execution Tracing** – You can test the execution of a State Table using the Step command to step through the current State Table one line at a time.

ASDL-to-State Table translation

At this stage of the translation process, the network element-specific processing is performed according to the ASDL that is being provisioned. The network element type and its software version are determined for the provisioning network element Host using configuration tables. ASAP uses these fields along with the ASDL command to find the State Table with the switch-specific information. When there are different network element Hosts, the command processor executes different State Tables.

Automatic State Table variables

You do not need to declare or define State Table variables since they are automatically created in State Table programs when they are used for the first time in a State Table. When a variable is assigned a value, that variable is created if it does not already exist. If the variable already exists, it is overwritten.

State Table extensibility

In addition to the action functions provided by the core libraries, an application such as a NEP or SRP can define action functions of its own, that can either overwrite or change the core action functions already provided. This ability to define and customize action functions, facilitates the encapsulation of provisioning-related activity into a single State Table action function, for use in State Table programs.

Loopback support

The State Table Interpreter supports the following modes of input/output operation:

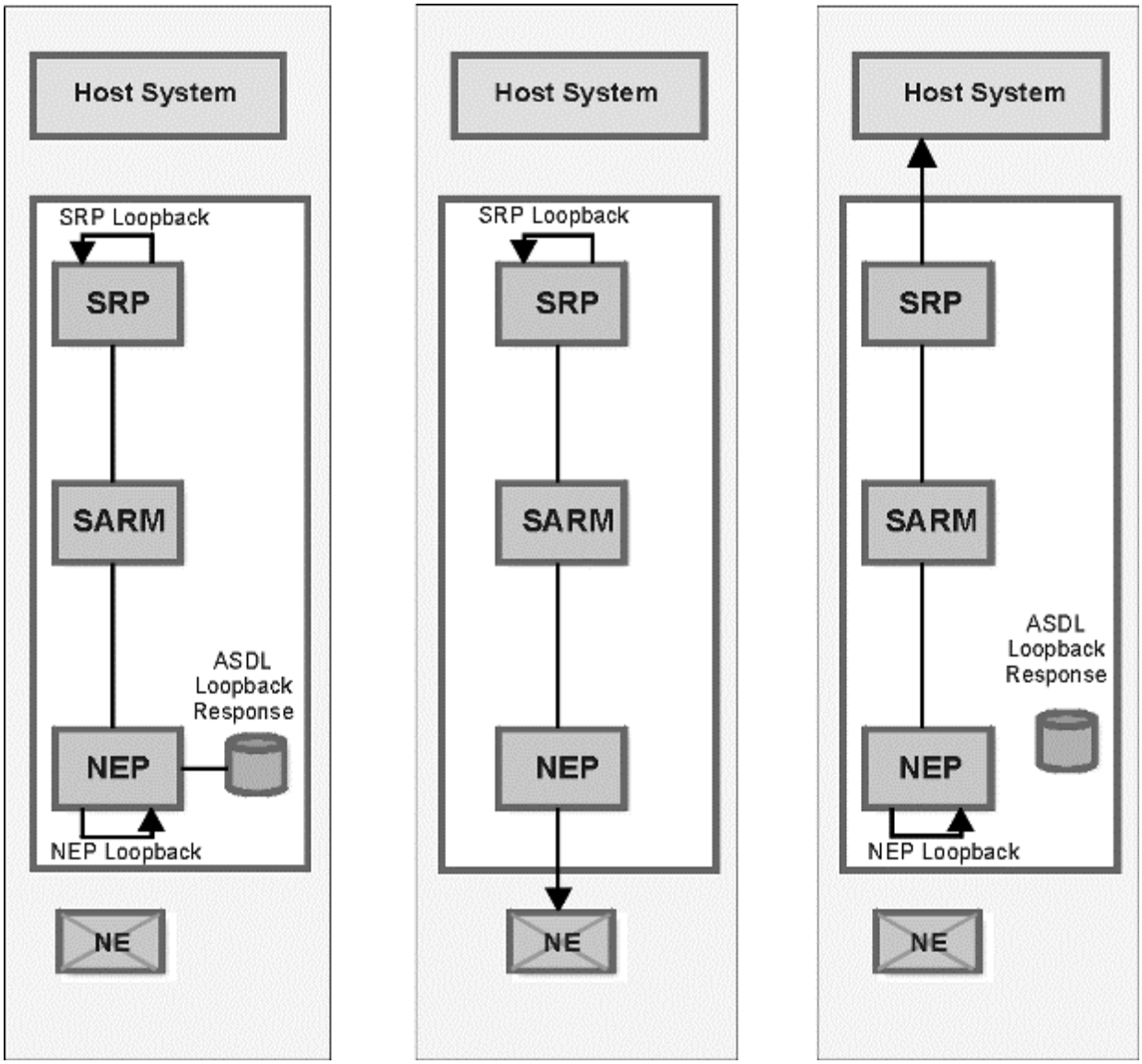
- **normal** – All input and output is transmitted to and received from the external system.
- **loopback** – All input and output operations are “looped around". This allows the testing of State Tables prior to interfacing with external systems. Any State Table action that performs input/output operations must provide its own loopback behavior.

You can test the ASAP interfaces (SRP to Host and NEP to NEs) independently of each other by running ASAP in one of the following loopback modes:

- Provide loopback at both the work order system and NE interfaces.
- Provide loopback at the work order system interface and valid connections to the NEs.
- Provide loopback at the NE interface and valid connections to the work order system.

Figure 5-3 outlines these loopback modes.

Figure 5-3 Loopback Modes



SRP loopback:

ASAP provides loopback at the work order system interface via an emulator-type application that can simulate an upstream system.

NEP loopback:

ASAP provides a test harness to the network elements by ensuring that as State Table action functions are written, they behave correctly when the NEP is put in loopback mode. Action functions that interact with network elements do not perform these actions while in loopback mode, but return success conditions instead. Therefore, you can configure the NEP in loopback mode by modifying a configuration parameter.

You can configure a static user-populated database table to determine the ASDL states that must be passed back to the SARM. Whenever an ASDL command is successfully completed in loopback mode, the State Table Interpreter in the NEP

references this table to determine the return status of the ASDL. This table allows all ASDL states to be passed back to the SARM and facilitates network element error condition testing within ASAP, even though ASAP is in loopback mode to the network elements.

Lexical Analysis Machine (LAM)

The State Table Interpreter includes a Lexical Analysis Machine (LAM) to parse complex network element reports from State Tables. The LAM is a set of action functions and data registers that allow the State Table developer to write State Tables that parse complicated data inputs, and extract formatted data for further processing or viewing.

Network element response parsing is the process of analyzing and creating data from the standard response provided by the network element. The LAM provides the State Table developer with a powerful tool for varying State Table execution based on network element responses.

Every operation that a State Table initiates updates a set of global registers within the State Table Interpreter. State Tables can access these registers through a pre-defined set of parameter names. Based on the content of the register, a State Table takes appropriate action

> **Note:**
>
> Only LAM actions can update global registers. All other actions use global registers in a read-only manner.

For more information on the LAM action functions, refer to the *ASAP Developer Reference*.

Database access from within State Tables

State Table programs can make calls to databases. These calls are generally made to SQL Server stored procedures which can perform static table lookups, dynamic table updates, etc. This allows you to create static and dynamic tables of provisioning-related information that is required in the provisioning process, and perform lookups and updates from within the State Table.

If data lookups are performed frequently, you can achieve better performance by loading the data into memory upon startup and writing a State Table action function to be called by the State Tables to perform a lookup of the in-memory copy.

Regular expression support

The core Interpreter action functions support the recognition of regular expressions which aid the parsing of information from external systems such as network elements.

Diagnostic and event support

The Interpreter provides State Table actions that support the writing of diagnostic messages to the server's diagnostic file and the raising of system events which can be mapped to system alarms from within the State Tables.

Customizing action functions

An action function is a C subroutine or a function that performs the associated State Table action. An action may have arguments called action strings and action integers. The action string of an action may contain no argument, one argument, or more than one argument. All arguments in the action string are separated by an "=" or ":". The action integer of an action can only be an integer. The following is an example format of an action in the State Table program.

```
CONCAT '%NEW_VAR=ABC_:%OLD_VAR' 0
```

The action function associated with this action retrieves all the arguments, concatenates ABC_ with the value of the variable %OLD_VAR, and then saves it in %NEW_VAR. After the execution of this action function, if not already defined, the NEW_VAR becomes a new variable retrievable by subsequent actions in the State Table program and chained State Table programs.

Writing action functions

Syntax:

The following is the syntax for an action function:

```
ACTION_STATUS action_func(CMD_PROC_DATA *data);
```

Input Parameters:

Each action function takes a pointer to the CMD_PROC_DATA structure as its argument. This data structure contains the following information:

- The ASDL parameters and variables generated in the State Table program before this action function is called.
- The communication variables for the device defined in the database table tbl_comm_param.
- The arguments passed to the action associated with this function in State Table program.

Return Values:

The action function returns ACTION_SUCCEED or ACTION_FAIL.

Using API routines

Use the ASAP API or database vendor routines to access databases.

Use system calls to access all resources available to the system. It is recommended that you check the ASAP and database vendor documents to find out whether similar routines are provided by the ASAP API and database vendor libraries.

Since the ASAP API is built on top of the database vendor and UNIX system libraries, and the database vendor API is built on the top of the UNIX system libraries, the ASAP API routine should be considered first, the database vendor API routine next, and the UNIX system routine last. For example, to move bytes from one location to another, the UNIX system call is memcpy(), the SYBASE API routine is srv_bmove() and the ASAP API routine is ASC_bmove() and should be used.

Retrieve arguments

Most actions accept one or more arguments in their action strings. The action function associated with the action should retrieve and check those arguments first.

Use CMD_get_assignment() to retrieve ASCII variables or use CMD_get_bvar_assignment() to retrieve binary variables. These two routines return the arguments to caller by the structures as below for CMD_get_assignment():

```
typedef struct cmd_assignment_arg {
     struct cmd_assignment_arg *next;
    CS_CHAR value[VAR_STR_L+1];
} CMD_ASSIGNMENT_ARG;
typedef struct {
    CS_CHAR label[VAR_NAME_L+1];
    CS_INT num_fields;
    CMD_ASSIGNMENT_ARG *arg_list;
} CMD_ASSIGNMENT_BUF;
for CMD_get_bvar_assignment():
typedef struct cmd_bvar_assignment_arg {
    struct cmd_bvar_assignment_arg *next;
    CS_INT len;
    CS_CHAR label[VAR_NAME_L+1];
    VOIDPTR value;
} CMD_BVAR_ASSIGNMENT_ARG;
typedef struct {
    CS_CHAR label[VAR_NAME_L+1];
    CS_INT num_fields;
    CMD_BVAR_ASSIGNMENT_ARG *arg_list;
} CMD_BVAR_ASSIGNMENT_BUF;
```

To expand the action string without delimiters (for example, SEND '%DMS_CMD' 5), use CMD_expand_action_string() to retrieve the argument passed to action.

Check the number of arguments passed to the action function (see sample program) to ensure that subsequent operations will be accomplished successfully.

Retrieving and storing parameters and variables

The ASDL parameters passed from the SARM and variables generated in the State Table program before the action function is called, may be accessed by the following routines. Once a variable is stored using CMD_store_var() and CMD_store_bvar(), it may be retrieved from this action function and the subsequent State Table program actions. Observe the following rules:

- Use CMD_get_var() to retrieve the parameters, ASCII variables and communication variables.
- Use CMD_store_var() to store the ASCII variables.
- Use CMD_get_bvar() to retrieve the binary variables.
- Use CMD_store_bvar() to store the binary variables.

Processing:

Call routines are provided by the ASAP, the database vendor, and UNIX library to process data.

Exit action function

An action function may exit in two conditions: ACTION_SUCCEED and ACTION_FAIL.

An action function must deallocate all spaces it has allocated before it exits. The routine CMD_free_assignment() is provided to deallocate the memory allocated by CMD_get_assignment().

Before an action function exits with a succeed status, it has to increase the variable PROGRAM_COUNTER by one in order to make the Interpreter proceed to the next action. The return statement will be:

```
return ACTION_SUCCEED;
```

When an action function exits with a failed status, the Interpreter will stop processing the rest of the actions in the State Table program. The return statement will be:

```
return ACTION_FAIL;
```

Action function example

```
static ACTION_STATUS get_field_func(CMD_PROC_DATA *data)
{
    CMD_ASSIGNMENT_BUF *assign_buf;

CMD_ASSIGNMENT_ARG *rec_label, *field_name, *rtn_data;
RECORD_LIST *rec_list;
CS_CHAR ascii[BINARY_TO_ASCII_L+1];

    DBBINARY *record;
    CS_INT len;
    /*
       The action associated with this function has format:

ACTION `%RTN=FIELD_A:%RTN_DATA`

    */
    /* Retrieve action arguments */
    if ((assign_buf = CMD_get_assignment(data)) == NULL) {
/* Log event and trigger alarm */

CMD_EVENT (CMD_DBG_INFO, "SYS_ERR", __LINE__, __FILE__,
"%d State Table Error: Can't Expand Action String [%s]",
SRQ_ID, CUR_ACT_STRING);
/* Stop processing the state table program */
return ACTION_FAIL;

    }
    /* Check whether the number of arguments is correct */
    if (assign_buf->num_fields != NUM_GET_FIELD_ARGS) {

CMD_DIAG(CMD_DBG_INFO, PROGRAM_LEVEL, "", __LINE__, __FILE__,
"Error: Not correct arguments for GET_FIELD");
CMD_free_assignment(assign_buf);
return ACTION_FAIL;

    }
    /* Pass arguments to variables which are easier to handle */
    rec_label = assign_buf->arg_list;
```

```
    field_name = rec_label->next;
    rtn_data = field_name->next;
    /* Pre-assign a return status to RTN */
    CMD_store_var(data, assign_buf->label, RTN_FAIL);
    /* Retrieve binary data saved under label rec_label->value */
    if (CMD_get_bvar(data, rec_label->value, (CS_VOID **)&record,

&len, (CS_VOID **)&rec_list)!= SUCCEED) {
CMD_EVENT (CMD_DBG_INFO, "SYS_ERR", __LINE__, __FILE__,
"Error: Can't Provide Get Binary Variable ");

    }
    /* process data - application specific */
    /* Save data with the label rtn_data->value */
    else if (CMD_store_var(data, rtn_data->value, ascii) == FAIL) {

CMD_EVENT(CMD_DBG_INFO, "SYS_ERR", __LINE__, __FILE__,
"Error: Can't store [%s]", ascii);

    }
    else
/* Since every thing is OK, set RTN to succeed */

CMD_store_var(data, assign_buf->label, RTN_SUCCEED);


   /* It mandatory to remove space allocated by CMD_get_assignment */
    CMD_free_assignment(assign_buf);
    /* Increase the program counter by 1 */
    PROGRAM_COUNTER++;
    /* Action function is done successfully */
    return ACTION_SUCCEED;
}
```

State Table Interpreter action functions

State Table Interpreter action functions are the core functions provided by the Interpreter library. Action functions control many aspects of state table operations and are used in most State Tables.

Interpreter Action Functions are used in all State Tables.

The core Interpreter Action Functions support the recognition of regular expressions that aid the parsing of information from external systems such as NEs.

A major feature of the Interpreter design is nested State Tables: the ability to call other State Table programs from the current one. This facilitates simpler, modular, and more compact State Table design. The State Table Interpreter supports the creation of libraries of State Table functions. Other State Tables can call these functions to perform well-defined tasks before returning to the calling State Table. This allows similar functionality to be collected in the same library.

Applications such as NEPs or SRPs can define their own action functions that either overwrite or change those already provided in the core libraries. This ability to define and customize action functions allows you to encapsulate provisioning-related activities into a single State Table action function.

- General action functions
- NEP action functions

General action functions

The following tables list State Table action functions and their equivalent JInterpreter actions and/or applicable classes. For more information on JInterpreter classes, refer to the *ASAP Online Reference*.

Table 5-26 General action functions

Action Function	Description	JInterpreter Action	Notes
#	Embeds comments in the State Table NPG file only.	Native	Use // or /** style comments in Java source files.
CALL	Calls a procedure.	Native	Invoke the Java class.method directly as a regular function call.
CASE, DEFAULT, SWITCH, ENDSWITCH	Makes decisions.	Native	Use Java **switch** statement.
CHAIN	-	Native	Invoke the Java class.method directly as a regular function call.
CLEAR	Manipulates variables.	Native	Assign null to the variable.
CMD_DUMP	Dumps the command processor's working data parameters.	-	Implement custom command dumping functionality using java.io classes.
COMMENT	Embeds comments in the State Table.	Native	Use // or /** style comments in Java source files.
CONCAT	Manipulates strings.	Native	Use Java String or Byte classes.
DECREMENT	Manipulates variables.	Native	Use arithmetic functionality in Java programming language.
DEF_REGEXPR	Defines a regular expression.	GNU Regular Expression library	Java-enabled NEP bundles a regular expression library that provides comparable functionality.
DIAG	Logs messages to the diagnostics/NE history.	com.mslv.activation.server Class Diagnostic	Class Diagnostic can be used to log messages to the diagnostic file.
EVENT	Issues system events.	com.mslv.activation.server Class EventLog	Class EventLog can be used to generate system events.
EXEC_RPC	Calls a database procedure.	-	Use JDBC library to invoke functions.

Table 5-26 (Cont.) General action functions

Action Function	Description	JInterpreter Action	Notes
EXIT	Deprecated.	-	-
EXPR_GOSUB	-	GNU Regular Expression library.	Java-enabled NEP bundles a regular expression library that provides comparable functionality.
FUNCTION	Defines a procedure.		Use Java language constructs to define functions.
GET_REGEXPR	Gets a regular expression	GNU Regular Expression library	Java-enabled NEP bundles a regular expression library that provides comparable functionality.
GOSUB	-	Native	Use Java language constructs to control program flow.
GOTO	Jumps to a new program statement with the indicated line number.	Native	Use Java language constructs to control program flow.
IF	-	Native	Use Java **if** statement.
IF_THEN, ELSE, ELSE_IF, ENDIF	Makes decisions.	Native	Use Java **if** statement.
IFDEF	-	com.mslv.activation.jinterpreter Class JProcessor	-
IFNDEF	-	getAllParams	The getAllParams methods returns a java.util.Properties object which can test for existence with the getProperty method.
INCREMENT	Manipulates variables.	Native	Use arithmetic functionality in Java programming language.
IND_SET	References complex compound variables (e.g. Arrays, Structures, etc.).	Native	Use String classes to concatenate values.
LENGTH	Manipulate strings	Native	Use String classes to concatenate and determine length values.
MAP_GOSUB	Maps a procedure.	-	Use Java Map interface and implementation classes to create and manipulate a custom map.
MAP_OPTION	Defines a map option.	-	Use Java Map interface and implementation classes to create and manipulate a custom map.
NEW_MAP	Defines a new map.	-	Use Java Map interface and implementation classes to create and manipulate a custom map.
RETURN	Exits a procedure and return to the calling State Table.	Native	Use Java language constructs to control program flow.

Table 5-26 (Cont.) General action functions

Action Function	Description	JInterpreter Action	Notes
SUBSTR	Manipulates strings	Native	Use Java String classes.
TRIM	Manipulates strings	Native	Use Java String classes.
WAIT	Pauses the program.	Native	Use java.lang.Thread.sleep.
WHILE, ENDWHILE	Creates loops.	Native	Use Java **while** statement.

For all general action functions, binary data types are handled as well as ASCII data types.

– Comment character

You can specify a comment that is ignored by the Interpreter while processing the State Table. This comment is stored in the database and is helpful to document particular State Table functionality.

See also "COMMENT."

Java equivalent – Use // or /** style comments in Java source files.

Syntax:

```
# Comment String
```

Example:

```
# This is a comment.
```

BCONCAT

Used for the concatenation of binary and ASCII variables into a binary variable.

See also "CONCAT," LENGTH," "SUBSTR," "TRIM."

The last colon in the action string is used to specify the concatenation of a blank terminated string.

Table 5-27 BCONCAT Parameters

Parameter	Description
%DEST	The destination variable name of the result of the concatenation.
a1, a2, .., an	Where A1 can be one of [%var | constant].

Syntax:

```
BCONCAT '%DEST=a1:a2:a3...:an:'
```

Example:

```
1000 CONCAT '%DEST1=ABC:DEF'
1010 CONCAT '%DEST2=%BinaryVar1:%AsciiVar1'
1020 BCONCAT '%DEST=%DEST1:%DEST2'
```

To embed the colon character with **BCONCAT**, use ASCII variable and **CONCAT** until "**:**" character is attached. When the colon character is attached at the end of a string, use **BCONCAT** to make the final destination variable a variable of the binary type.

The SOLUTION for **BCONCAT** concatenation of two arguments with a colon character between is:

```
1000 CONCAT '%VAR10=ABC'
1001 CONCAT '%VAR11=DEF'
1002 CONCAT '%VAR12=%VAR10:%;:'
1003 BCONCAT '%FINAL=%VAR12:%VAR11'
```

Therefore, the final binary destination variable (FINAL) is a concatenation of two arguments with a colon character between them (ABC:DEF).

CALC

Evaluates the arithmetic expression in the action string and saves the numerical value in the result variable. The arithmetic expression can be formed with decimal number, specified as literal or string variables, and the following operators:

- +
- -
- *
- /
- (
-)

The order of preference of the above variables is:

- **(** and **)**
- ***** and **/**
- **+** and **-**

The precision and scale of the output depends on the input. For example, the result of the expression (12.54 + 99.555) is 122.095. The result for the expression (555.55*-66) is -36666.30.

The result variable must be an ASAP Scalar variable and cannot be a Compound parameter. If no result variable exists, it is created and the result of the expression is saved in the variable. Any earlier value of a result variable is overwritten with the result of the expression.

Errors:

If the expression cannot be evaluated, due to mismatched brackets, undefined variables, etc., the action fails. This causes the associated ASDL to fail with an ASDL_STATE_TABLE_ERR status. The action function issues a SYS_ERR event.

Syntax:

```
CALC '%RES=<arithmetic expression>'
```

Example:

```
CALC '%RES=(%a * ((%b / %d) + %e))'
```

CALL

This action function executes a specific function within a State Table. The State Table is identified by the ASDL command to be executed, as well as the technology and software load of the NE to be provisioned. This indirect mapping (defined in tbl_nep_asdl_prog) is necessary to ensure that the correct version of the function is executed.

See also "CHAIN," "FUNCTION," "RETURN."

This action uses the **asdl_cmd**, technology, and software load to determine the State Table where this function resides. It then executes this function.

To use this function, the translation tables must be configured appropriately. In particular, the **asdl_cmd** must have a mapping to the relevant state table.

Java equivalent – Invoke the Java class.method directly as a regular function call.

Syntax:

```
CALL asdl_cmd::name
```

Example:

```
BEGIN EXAMPLE_1
#
# Call a function from a State Table utilities library and
# then chain the main processing ASDL.
#
1000 CALL 'UTILS_LIBRARY::RESET'
1010 CHAIN 'PROCESS_DATA'
1020 ASDL_EXIT 'SUCCEED'
END EXAMPLE_1
BEGIN UTILS_LIBRARY
#
# Library Function to reset variables
#
1000 FUNCTION 'RESET'
1010 CLEAR '%TEST_VAR'
1020 RETURN ''
END UTILS_LIBRARY
BEGIN PROCESS_DATA
#
1000 IF_THEN '%TEST_VAR DEFINED'
1010 CONCAT '%RESULT=%TEST_VAR'
1020 COMMENT 'Do other processing'
1030 ENDIF ''
1040 RETURN ''
#
END PROCESS_DATA
```

CASE

Checks a value against the current **SWITCH** value. If the values are equal, a subroutine call is made to the specified function address at the action integer.

See also "DEFAULT," "SWITCH," "ENDSWITCH."

Java equivalent – Use Java *switch* statement.

Syntax:

```
CASE <expand string> function address
```

Example:

```
1000 SWITCH '%TMP'
1010 CASE 'TEST' 2000
1020 CASE '%VAR1' 2500
1030 DEFAULT ''5000
1040 ENDSWITCH ''
1050 ASDL_EXIT 'SUCCEED'
# Handle TEST Case
2000 CONCAT '%TMP1=1'
2010 RETURN ''
# Handle Dynamic match of "%VAR1" and "%TMP"
2500 CONCAT '%TMP2=1'
2510 RETURN ''
# Handle Default case of no match
5000 CONCAT '%TMP3=1'
5010 RETURN ''
```

CHAIN

This action function is expanded to determine the ASDL command. The ASDL command is then used to determine which state table the Interpreter invokes.

See also "CALL," "FUNCTION," "RETURN."

It also uses the ASDL command in the expand string, technology, and software load to determine which state table to call. It then executes the state table.

The current state table is suspended until the chained state table completes.

To use the function, **CHAIN**, the translation tables must be configured appropriately. In particular, the ASDL must have a mapping to the relevant state table.

Java equivalent – Invoke the Java class.method directly as a regular function call.

Syntax:

```
CHAIN <expand string>
```

Example:

```
1000 CHAIN 'DMS_CHECK'
```

CLEAR

Sets the specified variable to NULL. It is the same as CONCAT “%variable=".

See also "CONCAT."

Java equivalent – Assign null to the variable.

Syntax:

```
CLEAR '%variable'
```

CMD_DUMP

This action function is expanded to determine the specified UNIX filename that is used to dump the Interpreter working data. This data includes all parameters currently set within the Interpreter.

Java equivalent – Implement custom command dumping functionality using java.io classes.

Syntax:

```
CMD_DUMP '<filename>'
```

Example:

Sample output appears below.

```
sunen214@caribou /u/itg/TEST/APIAF/EXECUTION >cat dump_file.doc
Command Processor FTP_DEV3 Data
Tech: DMS Software Version: BCS33 ASDL Command: A_CMD_DUMP_1 Port State: 1
Error       :
ASDL Status : 0
Response    :
Param Group : Undefined
Switch Value:
Variable Count: 0 Stack Depth 0 Program Counter 0
SARM Notify Needed: 1Port: 400618b0 Message: 406bfd58 Stack: 406af548 Var_tbl:
40919e70
Program Tbl: 4068f4b8, Current_Prog: 4068f4b8
Port Bind Dump
Device: FTP_DEV3 Pool SSTNDMS Line_type: F Command Prod Queue: 51
Vs Key: 0 Disabled: 0 Binded: 1 VS Qid: 56 Host: SSTNNBSCDS1 Sess Qid: 50

Variable Dump
Level 2 ASDL_CMD = [A_CMD_DUMP_1]
Level 2 DEVICE = [FTP_DEV3]
Level 1 DIAL_NO = []
Level 2 HOSTCLLI = [SSTNNBSCDS1]
Level 2 HOST_IPADDR = [<IP address>]
Level 2 HOST_NAME = [supra]
Level 0 HOST_PASSWORD = [<password>]
Level 2 HOST_USERID = [rtcenv20]
Level 1 IS_ROLLBACK = [NO]
Level 2 LOOPBACK_ON = [1]
Level 0 MCLI = [SSTNNBSCDS1]
Level 2 PORT = [21]
Level 2 SOFTWARE = [BCS33]
Level 2 SRQ_ID = [9]
Level 1 TECH = [DMS]
Level 2 WO_ID = [WO_CMD_DUMP_1]

Stack Trace

Active Program Dump
4068f4b8 A_CMD_DUMP_1 Action Tbl 406cf880 Count 2 Next 00000000
```

CMPND_COPY

Copies all parameters from one Compound parameter to another. The parameter to the right of the equal sign must be defined and must be a Compound parameter. In the example, OLD.a and OLD.b are copied into the new parameters NEWER.a and NEWER.b respectively.

Errors:

If the source parameter is undefined or is not a Compound parameter, the action fails. The failure of the action causes the associated ASDL to fail with an ASDL_STATE_TABLE_ERR status. The action function also issues a SYS_ERR system event.

Syntax:

```
CMPND_COPY '%variable=%variable'
```

Example:

CMPND_COPY ‘%NEWER=%OLD'

COMMENT

Specifies a comment that is ignored by the Interpreter.

See also "# – Comment character."

Java equivalent – Use // or /** style comments in Java source files.

Syntax:

```
COMMENT 'comment string'
```

Example:

```
1000 COMMENT 'This is a comment'
```

CONCAT

Concatenates the arguments into a value and then sets the destination variable to the new value. When it is necessary to append spaces to the end of a string, the action argument must be terminated with a colon. For example:

CONCAT ‘a1:a2: :'

See also "BCONCAT," "LENGTH," "SUBSTR," "TRIM."

> **Note:**
>
> There is a size constraint for the concatenated string by the "CONCAT" action function. When the combined string length of the three possible sources is greater than 255, a system event "SYS_ERR" is generated.

Java equivalent – Use Java String or Byte classes.

Syntax:

```
CONCAT %dest=a1:a2:a3:
```

Parameters:

- **%dest:** The destination variable name of the result of the concatenation.
- **a1:a2:a3:** Where a1 can be one of [%var | constant]. a2 and a3 are optional.

Example:

```
1000 CONCAT %TMP=ABC: :DEF
1010 CONCAT %TMP2=# Equivalent to CLEAR
1020 CONCAT %TMP3=%TMP:%TMP2:At the end
1030 CONCAT %TMP4=A:%;:B #results in A:B
```

The following table describes which built-in variables create reserved characters.

Table 5-28 Variables to create reserved characters

To insert	Use	Or
;	%COLON	%;
<space>	%SPACE	%<space>
=	%EQUAL	%=
%	%PERCENT	%%

COPY_TO_ASCII

Copies a state table binary variable value to an ASCII variable.

Syntax:

```
COPY_TO_ASCII     '%Ret=%Bvar[:If_Conversion]'
```

Parameters:

- **If_Conversion:** If the value is **0**, no conversion is done and source (Bvar) is copied to destination (Ret), as is (Binary copy).

  If the value is **1**, the source is converted to ascii values and then copied to the destination (only printable characters are copied, and the rest are converted to "**.**")

Remarks:

- **%BVar** must be a binary variable.
- No restrictions are imposed on the size of **%BVar** value.
- If %Bvar value is truncated, the action function sets the State Table variable %ASC_INFO_VAR to "TRUNCATED"; otherwise, it sets it to its default value (the empty string). The variable %ASC_INFO_VAR may be used to test for the condition if the value has been truncated.

Errors:

In the event of any of the following errors, state table program execution fails immediately.

- Number of arguments is less than (**%BVar** is missing). The following error message is printed: ‘Missing Mandatory Parameters'.

- **%BVar** does not exist or is not a binary variable. The following error message is printed: 'Var %BVar does not exist or is not a binary variable'.

Example:

```
100 BCONCAT '%N=1234'
110 COPY_TO_ASCII '%N1=%N'
120 COPY_TO_ASCII '%N2=%N:0'
130 COPY_TO_ASCII '%n3=%N:1'
140 IF_THEN '%ASC_INFO_VAR == 'TRUNCATED'
150 CONCAT '%I=0'
160 END_IF ''
```

DECREMENT

Decrements a variable within the State Table program by the value specified. If you do not specify a value, a default value of 1 is used.

See also INCREMENT.

Java equivalent – Use arithmetic functionality in Java programming language.

Syntax:

```
DECREMENT '%var' value
```

Example:

```
# Decrement INDEX by 1
1000 DECREMENT '%INDEX'
# Decrement VALUE by 5
1010 DECREMENT '%VALUE' 5
```

DEF_REGEXPR

Defines a regular expression that can be matched with a call to EXPR_GOSUB.

See also "EXPR_GOSUB."

You can have many regular expressions defined and then use EXPR_GOSUB to call the function of the first match it finds. This is similar to the UNIX **awk** utility.

The term **regular expression** is used here in the same general sense as UNIX uses the term. For more information, refer to **regexp** in the UNIX documentation.

Regular expressions are stored in a linked list. When a new regular expression is added, it is appended to the end of the list. Traversal of the list is performed from the head of the list until a match is found.

Syntax:

```
DEF_REGEXPR 'regular expression' function address
```

Example:

```
1000 DEF_REGEXPR '[A-Z].[A-Z]' 2000 # (e.g."A.A" or "ABC")
1010 EXPR_GOSUB '%BUF' 5000
# If value of %BUF is 'C1F', EXPR_GOSUB will set program counter to 2000.
# If value of %BUF is null, EXPR_GOSUB will set program counter to 5000.
```

DEFAULT

This action is a default processing action for a SWITCH statement when none of the cases is satisfied.

See also "ENDSWITCH," "SWITCH," "CASE."

If none of the cases you specify can be satisfied, a subroutine call is made to the function address that you specify.

Java equivalent – Use Java *switch* statement.

Syntax:

```
DEFAULT ` ' function address
```

Example:

```
1000 SWITCH `%TMP'
1010 CASE `TEST' 2000
1020 CASE `%VAR1' 2500
1030 DEFAULT `` 5000
1040 ENDSWITCH ``
1050 ASDL_EXIT `SUCCEED'
# Handle TEST Case
2000 CONCAT `%TMP1=1'
2010 RETURN ``
# Handle Dynamic match of "%VAR1" and "%TMP"
2500 CONCAT `%TMP2=1'
2510 RETURN ``
# Handle Default case of no match
5000 CONCAT `%TMP3=1'
5010 RETURN ``
```

DEL_REGEXPR

Deletes the specified regular expression from the list of regular expressions. If the action string is empty, all regular expressions are deleted from the list.

See also "DEF_REGEXPR," "EXPR_GOSUB."

The term **regular expression** is used here in the same general sense as UNIX uses the term.

For more information, refer to **regexp** in the UNIX documentation.

Syntax:

```
DEL_REGEXPR `regular expression'
```

Example:

```
# Delete the regular expression `*.AB*'
1000 DEL_REGEXPR `*.AB*'
# Delete all regular expressions
1010 DEL_REGEXPR ``
```

DIAG

Writes the expanded action string to the application's diagnostic file as a message with diagnostic level **DiagLevel**. If **DiagLevel:** is skipped, the level is defaulted to LOW_LEVEL. The diagnostic string can have '**:**' within it. This action is useful for cases in which you want to log a message only to the diagnostic file, not to the database.

See also "EVENT."

Java equivalent – com.mslv.activation.server Class Diagnostic. The Class diagnostic can be used to log messages to the diagnostic file.

Syntax:

```
DIAG 'DiagLevel:Diagnostic string'
```

Example:

```
1000 DIAG 'LOW:Invalid Remote NE %MCLI'
# Lines 1000 and 1010 are equivalent.
1010 DIAG 'Invalid Remote NE %MCLI'
1020 DIAG 'SANE:NE: %MCLI SRQ: %SRQID'
```

ELSE

Defines the section of the state table that is executed if a previous **IF** or **ELSE_IF** cannot be executed.

See also "IF_THEN," "ELSE_IF," "ENDIF."

This action is a logical construct that must match with corresponding **IF_THEN** or **ELSE_IF**, and **ENDIF** constructs.

To accomplish the **jump** to the end of the **IF_THEN** clause, the command processor inserts the reserved action **__GOTO_ENDIF** just before each **ELSE**. If you are debugging State Tables, this statement is apparent.

Java equivalent – Use Java if statement.

Syntax:

ELSE ‘‘

Example:

```
1000 IF_THEN '%TMP == "1"'
1010 LOG 'Test # 1 Complete'
1020 ELSE_IF '%TMP == "2"'
1030 LOG 'Test # 2 Complete'
1040 ELSE ''
1050 LOG 'Unknown Test Completion'
1060 ENDIF ''
```

ELSE_IF

If the expression specified is **True**, the next state table instruction is executed. Otherwise, execution continues at the next **ELSE_IF**, **ELSE**, or **ENDIF** state table instruction.

See also "IF_THEN," "ELSE," "ENDIF."

This logical construct must match with corresponding IF and ENDIF constructs.

To accomplish the **jump** to the end of the IF clause, the command processor inserts the reserved action **__GOTO_ENDIF** just before each **ELSE**. If you are debugging state tables, this statement becomes apparent.

Java equivalent – Use Java if statement.

Syntax:

```
ELSE_IF 'expression'
```

Example:

```
1000 IF_THEN '%TMP == "1"'
1010 LOG 'Test # 1 Complete'
1020 ELSE_IF '%TMP == "2"'
1030 LOG 'Test # 2 Complete'
1040 ELSE ''
1050 LOG 'Unknown Test Completion'
1060 ENDIF ''
```

ENDIF

Defines the end of a block of IF_THEN, ELSE, and ELSE_IF.

See also "IF_THEN," "ELSE," "ELSE_IF."

This logical construct must match the corresponding IF or IF_THEN constructs.

Java equivalent – Use Java if statement.

Syntax:

```
ENDIF ''
```

Example:

```
1000 IF_THEN '%TMP == "1"'
1010 LOG 'Test # 1 Complete'
1020 ELSE_IF '%TMP == "2"'
1030 LOG 'Test # 2 Complete'
1040 ELSE ''
1050 LOG 'Unknown Test Completion'
1060 ENDIF ''
```

ENDSWITCH

Marks the end of the SWITCH statement. A subroutine call made as a result of a CASE or DEFAULT statement returns to this point.

See also "CASE," "DEFAULT," "SWITCH."

This logical construct must match with a corresponding SWITCH construct.

Java equivalent – Use Java *switch* statement.

Syntax:

```
ENDSWITCH ''
```

Example:

```
1000 SWITCH '%TMP'
1010 CASE 'TEST' 2000
1020 CASE '%VAR1' 2500
1030 DEFAULT '' 5000
1040 ENDSWITCH ''
1050 ASDL_EXIT 'SUCCEED'
# Handle TEST Case
2000 CONCAT '%TMP1=1'
2010 RETURN ''
# Handle Dynamic match of "%VAR1" and "%TMP"
2500 CONCAT '%TMP2=1'
2510 RETURN ''
# Handle Default case of no match
5000 CONCAT '%TMP3=1'
5010 RETURN ''
```

ENDWHILE

Marks the end of a WHILE loop. Nested WHILE loops are not supported.

See also "WHILE."

Java equivalent – Use Java while statement.

Syntax:

```
ENDWHILE ''
```

Example:

```
1000 CONCAT '%I=1'
1010 WHILE '%I != "10"'
1020 LOG 'Loop iteration %I'
1030 INCREMENT '%I'
1040 ENDWHILE ''
```

ERROR_STATUS

Deprecated. Use "ASDL_EXIT."

EVENT

You can log ASAP system events from inside a State Table.

See also "DIAG."

Java equivalent – com.mslv.activation.server Class EventLog. Class EventLog can be used to generate system events.

Syntax:

```
EVENT 'Event:Event String'
```

The **Event** parameter must be the name of an event (maximum 8 characters) defined in the Control server, and the **Event String** parameter is used as the event text for the event (maximum 80 characters).

For more information on configuring events, refer to the *ASAP System Administrator's Guide*.

EXEC

Deprecated. This action function is provided for backward compatibility only. Use "EXEC_RPC."

EXEC_RPC

This is a more general purpose action than **EXEC** and must be used in any new State Table development instead of **EXEC**.

This action executes the function in the NEP database (**NEP_USER** and **NEP_PASSWORD** are used to determine the source of the **STORED_PROC**) as follows:

```
EXEC_RPC STORED_PROC @asdl="Current ASDL Command",

@tech = "Technology",
@sftwr_load = "Software Load",
@arg1 = arg1, ... @argn = argn
```

Only **arg1**, **arg2**, etc. arguments are acceptable.

Multiple columns and rows of data can be returned by the function. The following rules are used to store the variables for later use:

- Each row increments an index value by 1 starting at 1.
- For the first row, the column labels are used to set up parameters in a structure, for example, var.column1, var.column2, var.column3.
- For each row returned, including the first row, the column labels are used in conjunction with the current index to create an array of structures, for example, var[%index].column1, var[%index].column2.

  If the column label is NULL or '', a normal array element is created, for example, var[%index] and "var" is also assigned the value. In this case, "%var" is set to the value in the last row where column label is NULL or ''.

  Examples:

  ```
  var[1].column1 and var.column1    = value at row 1, column 1
  var[1].column2 and var.column2    = value at row 1, column 2
  var[1] and var = value at row 1, column 3 # if column 3 has no label
  var[2].column1 = value at row 2, column 1
  var[2].column2 = value at row 2, column 2
  var[2] = value at row 2, column 3
  ```

This function can pass many parameters to the stored function. It can also retrieve many data rows and columns back from the stored function. These data rows and columns are stored as indexed parameters within the Interpreter.

The EXEC_RPC function can retrieve only one result set specified with a cursor variable in the stored function.

The stored functions to execute via EXEC_RPC should be defined as functions with the following first four parameters mandatory:

```
cursor_var CURSOR
asdl CHAR
```

```
tech CHAR
sftwr_load CHAR
```

Converting float data types can cause rounding errors. All data types are converted into characters.

Java equivalent – Use JDBC library to invoke functions.

Syntax:

```
EXEC_RPC '%var=STORED_PROC:arg1:.:argn'
```

Example:

```
State table 1:
1000 EXEC_RPC '%X=SSP_get_dn_list:%MACH_CLLI:%LEN'
# dir_no dn_type
# 2531920SINGLE PARTY LINE
# 2531921MULTIPLE APPEARANCE DIRECTORY NUMBER
#
# Results:
# X[1].dir_no = "2531920",
# X[1].dn_type = "SINGLE PARTY LINE",
# X[2].dir_no = "2531921",
# X[2].dn_type = "MULTIPLE APPEARANCE DIRECTORY NUMBER"


State Table 2:
1000 EXEC_RPC '%X=SSP_get_dn_list:%HOST_CLLI:%SITE:%FROM_LEN'
# X[1] = MCLI or 'UNKNOWN'



Stored function:
TYPE SSP_get_mcli_rt1 IS RECORD (
        tmp             VARCHAR2(255)
 );

TYPE SSP_get_mcli_1 IS REF CURSOR RETURN SSP_get_mcli_rt1;

CREATE OR REPLACE FUNCTION SSP_get_mcli(
RC1     IN OUT NepPkg.SSP_get_mcli_1,
asdl            CHAR ,
tech            CHAR ,
sftwr_load      CHAR ,
arg1            tbl_clli_len_ltg.host_clli%TYPE,
arg2            tbl_clli_len_ltg.site%TYPE,
arg3            tbl_clli_len_ltg.from_len%TYPE)
RETURN INTEGER
AS
StoO_selcnt     INTEGER;
StoO_error      INTEGER;
StoO_rowcnt     INTEGER;
StoO_errmsg     VARCHAR2(255);
tmp             VARCHAR2(128);
BEGIN
        BEGIN
                StoO_rowcnt := 0;
                StoO_selcnt := 0;
                StoO_error  := 0;
                SELECT   mach_clli
                INTO SSP_get_mcli.tmp
```

```
                FROM tbl_clli_len_ltg
                WHERE host_clli = SSP_get_mcli.arg1
                AND site = SSP_get_mcli.arg2
                AND SSP_get_mcli.arg3 >= from_len
                AND SSP_get_mcli.arg3 <= to_len;
                StoO_rowcnt := SQL%ROWCOUNT;
                EXCEPTION
                        WHEN OTHERS THEN
                                StoO_rowcnt := 0;
                                StoO_selcnt := 0;
                                StoO_error := SQLCODE;
        END;
        IF StoO_rowcnt = 0 THEN
                SSP_get_mcli.tmp :=  'UNKNOWN';
        END IF;
        StoO_rowcnt := 0;
        StoO_selcnt := 0;
        StoO_error  := 0;
        OPEN RC1 FOR
        SELECT  SSP_get_mcli.tmp
        FROM DUAL;
RETURN 0;
END
```

EXEC_RPCV

To improve the efficiency of database queries conducted by ASAP, the EXEC_RPC action function has been split into two groups: EXEC_RPC is used for database update/insert functionality and has a cursor and EXEC_RPCV is used for database queries and employs no cursor.

Syntax:

```
    EXEC_RPCV   '%DEST=SSP:arg1:.:argn'
```

where…

- DEST is the name of the return variable that holds the result from the stored procedure.
- SSP is the name of the stored procedure in the NEP database to execute.
- arg1...argn are the input paramters provided for use in the stored procedure.

Example (usage in a state table):

```
BEGIN ST1
......
100          EXEC_RPCV   '%R1=SSP_test:1:35'
110          LOG 'Stored procedure returned %R1'
......
500          ASDL_EXIT 'SUCCEED'
END ST1
```

In this sample, R1 holds the result. %R1 will log the result into tbl_srq_log.

The stored procedure declaration and definition for EXEC_RPCV is similar to EXEC_RPC. The difference is that the first parameter declared in stored procedures for EXEC_RPCV is an OUT (output variable) instead of a cursor variable.

The following is an example of a stored procedure in the NEP database:

```
CREATE OR REPLACE FUNCTION SSP_test(
retv    OUT VARCHAR2 ,
asdl            CHAR ,
tech            CHAR ,
sftwr_load      CHAR ,
op              VARCHAR2 ,
data            VARCHAR2)
RETURN INTEGER
AS
StoO_selcnt     INTEGER;
StoO_error      INTEGER;
StoO_rowcnt     INTEGER;
StoO_errmsg     VARCHAR2(255);
BEGIN
        StoO_rowcnt := 0;
        StoO_selcnt := 0;
        StoO_error  := 0;
        SSP_test.retv := TO_NUMBER(SSP_test.op) || TO_NUMBER(SSP_test.data);
        RETURN 0;
END SSP_test;
/
```

This sample stored procedure has the following parameters:

- **retv OUT VARCHAR2:** This return parameter will return the result (see R1 state table variable in the state table above). This should less than 256 characters.
- **asdl CHAR:** ASDL name (this is not provided through EXEC_RPCV state table action function, but passed internally to the stored procedure).
- **tech CHAR:** Technology (this is not provided through EXEC_RPCV state table action function, but passed internally to the stored procedure).
- **sftwr_load CHAR:** Software load (this is not provided through EXEC_RPCV state table action function, but passed internally to the stored procedure).
- **op VARCHAR2:** Optional input parameter 1.
- **data VARCHAR2:** Optional input parameter 2.

The first four parameters are mandatory. After these four parameters, optional parameters should be declared.

Note that the return result is assigned to the variable SSP_test.retv so that it can be retreived via %R1.

You do not need to provide optional parameters. In this case, the stored function appears as follows:

```
CREATE OR REPLACE FUNCTION SSP_test1(
retv    OUT VARCHAR2 ,
asdl            CHAR ,
tech            CHAR ,
sftwr_load      CHAR)
RETURN INTEGER
AS
StoO_selcnt     INTEGER;
StoO_error      INTEGER;
StoO_rowcnt     INTEGER;
StoO_errmsg     VARCHAR2(255);
BEGIN
        StoO_rowcnt := 0;
        StoO_selcnt := 0;
```

```
        StoO_error  := 0;
        SSP_test1.retv := 'This is the return result from SSP_test1 stored
function.';
        RETURN 0;
END SSP_test1;
/
Only the mandatory parameters are declared. There are no optional parameters. In
a State Table, EXEC_RPCV is executed as follows for this stored function:
BEGIN ST2
......
100          EXEC_RPCV   '%R2=SSP_test1'
110          LOG 'Stored procedure returned %R2'
......
500          ASDL_EXIT 'SUCCEED'
END ST2
```

The R2 variable in this State Table will have the following string after EXEC_RPCV is executed:

"This is the return result from SSP_test1 stored function".

EXIT

Deprecated. Use "ASDL_EXIT."

EXPR_GOSUB

The buffer is checked against all the regular expressions that have been defined using **DEF_REGEXPR**. If a match is found, a **gosub** is executed to the function specified by the expression. If no expression is found, the default function specified by this action is used to execute a **gosub**.

See also "GOSUB," "GOTO," "MAP_GOSUB," "RETURN."

Syntax:

```
EXPR_GOSUB 'buf' function address
```

Parameters:

- **buf:** The buffer to be checked.
- **function address:** The default function address if no match is found in the buffer.

Example:

```
1000 DEF_REGEXPR '[A-Z].[A-Z].[0-9]' 2000 # (e.g. "A.AB3
or C1D77")
1030 EXPR_GOSUB ' %BUF' 5000
# If value of %BUF is 'S.F.9', for example, EXPR_GOSUB will set program counter
to 2000.
# If value of %BUF is null, EXPR_GOSUB will set program counter to 5000.
```

FUNCTION

Defines a function in a State Table program. The function can be defined as part of a library or locally as part of the State Table program from which it is invoked. If invoked locally, the function must be defined after all the invocations.

See also "CALL," "CHAIN," "RETURN."

Syntax:

```
FUNCTION 'name'
```

Example:

```
BEGIN EXAMPLE_1
#
# Call a function from a State Table utilities library and
# then chain the main processing ASDL.
#
1000 CALL 'UTILS_LIBRARY::RESET'
1010 CHAIN 'PROCESS_DATA'
1020 asdl_EXIT 'SUCCEED'
END EXAMPLE_1
BEGIN UTILS_LIBRARY
#
# Library Function to reset variables
#
1000 FUNCTION 'RESET'
1010 CLEAR '%TEST_VAR'
1020 RETURN ''
END UTILS_LIBRARY
BEGIN PROCESS_DATA
#
1000 IF_THEN '%TEST_VAR DEFINED'
1010 CONCAT '%RESULT=%TEST_VAR'
1030 ENDIF ''
1040 RETURN ''
#
END PROCESS_DATA
```

GET_REGEXPR

Retrieves an array of regular expressions (specified by **expr** from the buffer (specified by **buf**).

See also "DEF_REGEXPR."

The array is built as follows: dest[1], dest[2], etc.

Syntax:

```
GET_REGEXPR '%dest=expr:buf'
```

Example:

```
1000 CONCAT '%BUF=HOST 00 1 10 23 NEWC 10 1 23 01'
1010 GET_REGEXPR '%SITE=[A-Z][A-Z][A-Z][A-Z]:%BUF'
#
# SITE[1]= HOST, SITE[2] = NEWC
```

GOSUB

Makes a subroutine call to the function specified by the function address.

See also "EXPR_GOSUB," "GOTO," "MAP_GOSUB," "RETURN."

Java equivalent – Use Java language constructs to control program flow.

Syntax:

GOSUB ‘ ’ function_address

Example:

```
1000 GOSUB '' 2000
# Continue Processing ...
#
# Subroutine
#
2000 CONCAT '%TMP=1'
2010 RETURN ''
```

GOTO

Goes to the function address you specify and continues executing.

> **Note:**
>
> Avoid using GOTO if possible. The use of GOTO can seriously impair the readability of and ability to debug a state table. Instead, use GOSUB, CHAIN or FUNCTION.

See also "EXPR_GOSUB," "GOSUB," "MAP_GOSUB," "RETURN."

Java equivalent – Use Java language constructs to control program flow.

Syntax:

```
GOTO ' ' function address
```

Example:

```
1000 GOTO '' 2000
1010 CONCAT '%TMP=Executed Second'
2000 CONCAT '%TMP=Executed First.'
2010 RETURN ''
```

IF

If the specified expression is **True**, a subroutine call is made to the specified line number. Otherwise, execution continues with the next state table instruction.

See also "IFDEF," "IFNDEF."

The **IF_THEN** action, similar to this action, does not perform a subroutine call.

Java equivalent – Use Java if statement.

Syntax:

```
IF 'expression' line #
```

Example:

```
1000 IF '%TMP >= "TEST"' 2000
# Continue Processing ...
2000 LOG 'This is Test %TMP'
2010 RETURN ''
```

IF_THEN

If the specified expression is **True**, the next state table instruction executes. Otherwise, execution continues at the next **ELSE_IF**, **ELSE** or **ENDIF** state table instruction.

See also "ELSE," "ELSE_IF," "ENDIF."

An expression is defined as follows:

```
operator::= { ==, !=, <=, >=, <, >, NOT_NULL, IS_NULL, DEFINED, NOT_DEFINED}
expression ::= %var operator [{ %var | number | "string"}]
```

The maximum number size must be within the boundary of double precision.

The State Table interpreter compares the values of integers or real numbers if the contents of the both value1 and value2 are numbers in the IF_THEN value1 REL_OP value2. An example of numbers in this format are 123, 12.3, +123, -123.

If using the IF_THEN clause to compare IP addresses, keep in mind that only IPv4 addresses are supported. The '= =' comparison for IP addresses behaves as follows:

- the format of the IP address should be a.b.c.d. Formats such as 'a.b.c', 'a.b', 'a', 'a..b.c' are not supported.
- leading 0s are considered to be an octet. For example, if the IP address is 010.9.12.1, then 010 is considered to be an octet and the actual IP address is interpreted as 8.9.12.1. In other words, 010.9.12.1 does not equal 10.9.12.1.
- all digits that have leading 0s should be less than 8 since it is recoginzed as an octet. 089.20.1.195 is a malformed IP address which is consequently not recognized as an IP address.
- 10.001.24.196 and 10.1.24.196 are recognized as same IP address. Although though second part of first IP address "001" is considered as an octet and is therefore "1".

> **Note:**
>
> If the IP address is not valid, it is considered to be a string.

Java equivalent – Use Java if statement.

Syntax:

```
IF_THEN 'expression'
```

Example:

```
1000 IF_THEN '%TMP == "1"'
1020 LOG 'Test # 1 Complete'
1030 ELSE_IF '%TMP == "2"'
1040 LOG 'Test # 2 Complete'
1050 ELSE ''
1060 LOG 'Unknown Test Completion'
1070 ENDIF ''
```

Example for IP addresses:

```
1000 CONCAT '%TMP' = 10.1.9.125
1010 IF_THEN '%TMP == "10.1.9.125"' (returns true)
...
2000 IF_THEN '%TMP == "010.1.9.125"' (returns false)
2010 IF_THEN '%TMP == "10.1.09.125"' (returns false)
```

IFDEF

Checks the variable that you specify. If it is defined, the action makes a subroutine call to the function that you specify.

See also "IF," "IFNDEF."

Java equivalent – com.mslv.activation.jinterpreter Class JProcessor

Syntax:

```
IFDEF '%var' function address
```

Example:

```
1000 IFDEF '%TMP' 2000
1010 ...
2000 LOG 'TMP Variable Defined.'
2010 RETURN ''
```

IFNDEF

Checks the variable that you specify. If it is not defined, the action makes a subroutine call to the function that you specify.

See also "IF," "IFDEF."

Java equivalent – getAllParams. The getAllParams methods returns a java.util.Properties object which can test for existence with the getProperty method.

Syntax:

```
IFNDEF '%var' function address
```

Example:

```
1000 IFNDEF '%TMP' 2000
1010 ...
2000 LOG 'TMP Variable Undefined.'
2010 RETURN ''
```

INCREMENT

Increments a variable within the state table program by the specified value.

See also "DECREMENT."

Java equivalent – Use arithmetic functionality in Java programming language.

Syntax:

```
INCREMENT '%var' value
```

Example:

```
1000 INCREMENT '%INDEX'
# Increment INDEX by 1
1010 INCREMENT '%VALUE' 5
# Increment INDEX by 5
```

IND_SET

Concatenates the parameters specified by p1, p2, and p3 to generate the name of the variable to be retrieved. For example, %OPT=OPT:%OPT_IDX can be used to generate OPT1, OPT2, ..., OPTn.

See also "CONCAT."

> **Note:**
>
> Do not use this action for any new state table development. This action is included for backward compatibility only. Any new state tables must use the extended variable syntax (in other words, %{...}).

Java equivalent – Use String classes to concatenate values.

Syntax:

```
IND_SET '%dest=p1:p2:p3'
```

Example:

```
1000 CONCAT   '%IDX=3'
1005  CONCAT  '%ARRAY[3]=abc'
#
# These two statements are equivalent.
#
1010  CONCAT  '%TMP=%{ARRAY[%IDX]}'
1020  IND_SET  '%TMP=ARRAY[:%IDX:]'
# Value of variable TMP will be: 'abc'.
```

LENGTH

Concatenates the parameters specified by p1, p2 and p3 and then sets the destination parameter to the length of the string (maximum 255 bytes). The optional parameters are p2 and p3.

See also "CONCAT," "SUBSTR," "TRIM."

Java equivalent – Use String classes to concatenate and determine length values.

Syntax:

```
LENGTH '%dest=p1:p2:p3'
```

Example:

```
1000  CONCAT     '%TMP1=abc'
1010  CONCAT      '%TMP2=defg'
1020    CONCAT  '%TMP3=hijklm'
1020  LENGTH  '%LEN1=%TMP1:%TMP2:XX'
```

```
1030  LENGTH  '%LEN2=12:%TMP3:21'
In this example, LEN1 = 9 and LEN2 = 10.
```

MAP_GOSUB

Searches the specified map to locate a value. If the value is found, the subroutine specified by the **MAP_OPTION** action is called. If the value is not found, the default subroutine specified when the map was created, for example, **NEW_MAP** action is executed.

See also "MAP_OPTION," "NEW_MAP," "RETURN."

Java equivalent – Use Java Map interface and implementation classes to create and manipulate a custom map.

Syntax:

```
MAP_GOSUB '%mapname "%value"'
```

Example:

```
1000  NEW_MAP  'PARSER_MAP'   2000
1010  MAP_OPTION  'PARSER_MAP "LEN:"'   2100
1020  MAP_OPTION  'PARSER_MAP "TYPE:"'   2200
1030  MAP_OPTION  'PARSER_MAP "SNPA:"'   2300
1040  MAP_GOSUB  'PARSER_MAP "%PARSE_BUF"'
```

In this example, if **%PARSE_BUF = "LEN"**, execution of the program continues from line 2100.

MAP_OPTION

Specifies the subroutine to call when the specified option value is used with the **MAP_GOSUB** function.

See also "MAP_GOSUB," "NEW_MAP."

Java equivalent – Use Java Map interface and implementation classes to create and manipulate a custom map.

Syntax:

```
MAP_OPTION '%mapname "%value"' function address
```

Example:

```
1000  NEW_MAP  'PARSER_MAP'   2000
1010  MAP_OPTION  'PARSER_MAP "LEN:"'   2100
1020  MAP_OPTION  'PARSER_MAP "TYPE:"'   2200
1030  MAP_OPTION  'PARSER_MAP "SNPA:"'   2300
1040  MAP_GOSUB  'PARSER_MAP "%PARSE_BUF"'
```

MASK

Concatenates variables A1, ... A(*n*) and then applies the mask to the resulting string.

- **source** – String to which MASK string is applied to.
- **destination** – The resulting string after MASK operation.
- **mask** – A string that modifies the source according to next rules:

If the mask includes the following characters:

- **x** – Ignores source character (delete). It does not appear in the destination string.
- **y** – Copies a character from the source to the destination string.
- Any other character – Inserts that character from MASK into the destination string.
- Use **\x** or **\y** – Forces (inserts) an **x** or **y** from MASK into the destination strings.

Syntax:

```
MASK'%DEST=%MASK:%A1:%A2:... :%An'
```

Example:

If the parse string is 'ABCDEF', the following MASKs produce the following results:

Table 5-29 Sample mask results

MASK	Results
yyyyyy	ABCDEF
yy.yy.yy	AB.CD.EF
xy.xy.xy	B.D.F
yx.yx.yx	A.C.E
XYZ-yy-\x\y\z	XYZ-AB-xyz
<yyy><yyy>	<ABC><DEF>
<yxy><xyx>	<AC><E>

Each **x** and **y** in the MASK (not including the ones prefixed with a "****") map directly to a character in the source string. Therefore, the 4th "**x**" or "**y**" in the MASK determines the processing on the 4th character in the source, and so on.

Incorrect:

```
"yy yy yy" => AB CD EF
```

Correct:

```
yy yy yy => AB CD EF
```

NEW_MAP

If a value cannot be located in the map options, this action defines a new map within the state table and the default subroutine to be executed.

See also "MAP_GOSUB," "MAP_OPTION."

Java equivalent – Use Java Map interface and implementation classes to create and manipulate a custom map.

```
NEW_MAP 'mapname' function address
```

Example:

```
1000  NEW_MAP  'PARSER_MAP'   2000
1010  MAP_OPTION  'PARSER_MAP "LEN:"'   2100
1020  MAP_OPTION  'PARSER_MAP "TYPE:"'   2200
```

```
1030  MAP_OPTION  'PARSER_MAP "SNPA:"'   2300
1040  MAP_GOSUB  'PARSER_MAP "%PARSE_BUF"'
```

PAD_CHAR

Allows the size of the variable **%VAR** to be expanded to the length, **LENGTH**. The new expanded area will be filled with the argument character, **CHARACTER**. The value of the variable is preserved if the size of the expanded variable is greater than the current size. Otherwise, an error occurs.

See also "ZERO_PAD."

Syntax:

```
PAD_CHAR '%RTN=%VAR:LENGTH:CHARACTER'
```

Example:

```
1000 BCONCAT   '%DEST=Sample1:Sample2'
1010 PAD_CHAR   '%RTN=%DEST:%Length:0'
```

PAUSE

Deprecated. Use "WAIT."

RETURN

Returns from a subroutine or state table chain.

See also "CALL," "CHAIN," "EXPR_GOSUB," "FUNCTION," "GOSUB," "MAP_GOSUB."

Java equvalent – Use Java language constructs to control program flow.

Syntax:

```
RETURN '  count
```

Parameters:

- **count:** Optional, default is **1**. If the count is set, multiple returns can be executed at once. A count of **0** is equivalent to a count of **1**.

> **Note:**
>
> The count value must be used with caution as this is the equivalent to using a GOTO by skipping return points on the stack.

Example:

```
1000  GOSUB  ''   2000
1010  Continue Processing ...
# Subroutine
#
2000  CONCAT  '%TMP=1'
2010  RETURN  ''
```

SUBSTR

Sets the destination variable to a substring of the source string of the length you specify and starting at the offset you specify.

See also "CONCAT," "LENGTH," "TRIM."

Java equivalent – Use Java String classes.

Syntax:

```
SUBSTR '%dest=%sr c:off:len'
```

Parameters:

- **%des t:** The destination string variable name.
- **%src:** The source string variable name.
- **off:** The offset at which to begin the substring. The first character of the source string is offset at 0.
- **len:** The length of the substring.

Example:

```
1000  CONCAT  '%TMP=1234567890'
1010  SUBSTR  '%TMP1=%TMP:0:2'  # %TMP1 = 12
1020  SUBSTR  '%TMP2=%TMP:9:5'  # %TMP2 = 0
```

SWITCH

Sets the current switch value for the state table.

See also "CASE," "DEFAULT," "ENDSWITCH."

The use of curly braces is highly recommended when fully expanding variable names that include non-alphanumeric characters (especially dot "**.**", left bracket **[**, or right bracket **]**) carrying special meaning.

Java equivalent – Use Java *switch* statement.

Syntax:

SWITCH '<expand string>'

Example:

```
1000  SWITCH  '%TMP'
1010    CASE  'TEST'  2000
1020    CASE  '%VAR1'  2500
1030    DEFAULT  ''  5000
1040  ENDSWITCH  ''
1050  ASDL_EXIT  'SUCCEED'
# Handle TEST Case
2000  CONCAT  '%TMP1=1'
2010  RETURN  ''
# Handle Dynamic match of "%VAR1" and "%TMP"
2500  CONCAT  '%TMP2=1'
2510  RETURN  ''
# Handle Default case of no match
```

```
5000  CONCAT  '%TMP3=1'
5010  RETURN  ''
```

TRIM

Concatenates the parameters p1, p2, and p3 into a larger parameter and then trims the parameter based on the **trim_flag**. The trimmed result is stored in the destination variable.

See also "CONCAT," "LENGTH," "SUBSTR."

Java equivalent – Use Java String classes.

Syntax:

```
TRIM '%dest=p1:p2:p3' trim_flag
```

Parameters:

- **%dest:** Destination variable.
- **p1, p2, p3:** Parameters passed to this action. Optional parameters are p2 and p3.
- **trim_flag:** Can be set to indicate whether or not leading blanks, trailing blanks, or both must be stripped. Possible values are:
  - **0** – No trimming
  - **1** – Leading blanks trim
  - **2** – Trailing blanks trim
  - **3** – Leading and trailing blanks trim

Example:

```
# Trim leading and trailing blanks from TMP
1000 TRIM  '%TMP=%TMP'  3
# Trim trailing blanks
1010  TRIM  '%TMP=%BUF:%XYZ'   2
```

WAIT

Pauses the State Table execution for the time you specify in seconds. If the time is **0**, it defaults to one second.

Java equivalent – Use java.lang.Thread.sleep.

Syntax:

```
CONCAT '%WTIME=15'
WAIT '%WTIME'
```

Example:

```
# WAIT State Table execution for 120 seconds.
1000 WAIT  '120  '
```

WHILE

Starts a WHILE loop. If the expression is “True", State Table execution continues with the next State Table operation. If the expression is not true, the State Table execution resumes at the point following the ENDWHILE statement.

See also "ENDWHILE."

Java equivalent – Use Java while statement.

Syntax:

```
WHILE 'expression'
```

Example:

```
1000  CONCAT  '%I=1'
1010  WHILE  '%I != "10"'
1020  LOG  'Loop iteration %I'
1030  INCREMENT   '%I'
1040  ENDWHILE  ''
```

ZERO_PAD

Specifies the state table variable label. The state table variable is retrieved and the numeric value is saved in the variable which is preceded with leading zeroes. The number of leading zeroes equals field length as specified - (minus) value length. For example:

See also "PAD_CHAR."

```
    ZERO_PAD '%NUM_VAR'  5
```

If the variable **%NUM_VAR** contains a numeric field **345**, **ZERO_PAD** action causes the value **00345** to be saved in the **NUM_VAR** variable. The number of leading zeroes is 2 in this case. If **NUM_VAR** contains a value of 3, **00003** is stored.

If **NUM_VAR** contains **1234567**, **ZERO_PAD** truncates the field to **12345**. The action integer specifies the total field length. If the action integer is set to **0**, the number of leading zeroes is defaulted to **1**.

> **Note:**
>
> To reset the parameter that ZERO_PAD has been applied to its original value, set the action integer to -1. For example: ZERO_PAD '%NUM_VAR' -1

Syntax:

```
ZERO_PAD 'variable label' field length
```

NEP action functions

NEP action functions relate primarily to switch history and parameter management tasks, such as:

- Data transmission and reception between ASAP and NEs.

- Virtual screen manipulation, if the communication to the network element is terminal-based
- NE access control
- NE response logging and switch history
- NE blackout management

NEP action functions support the manipulation and transmission of all types and formats of parameters, including indexed and compound parameters.

You can change the core action functions that are provided in the NEP library or overwrite the existing ones as required.

The sample NEP provides some additional functions that perform NE-specific activities such as lookups from static tables and data formatting.

Other NEP action functions provide functions related to the transmission of network element responses and parameters from the NEP to the SARM, as well as some statistical functionality.

The following NEP action functions relate to the transmission of NE responses and parameters back from the NEP to the SARM and also offer some statistical functionality.

Table 5-30 NEP Action Functions

Action Function	Description	Java Method	Notes
LOG	Log messages to the NE history.	log	JProcessor class. Generate an NE history log item.
PARAM_GROUP	Set parameter grouping field.	setParamGroup	JProcessor class. Sets a group identifier.
SEND_COMPND SEND_PARAM	Send parameters to SARM.	returnCSDLParam returnCompoundCSDLParam returnGlobalParam returnCompoundGlobalParam returnInfoParam returnCompoundInfoParam returnRollbackParam returnCompoundRollbackParam	JProcessor class. Return parameters up to SARM.
ASDL_EXIT	Exit the program specifying the ASDL_EXIT status.	setASDLExitType	Set exit type and return from the function.

The following table describes ASAP NEP action functions that manage the following:

- The data transmission and reception between ASAP and the NE.
- Virtual screen manipulation, if the communication to the NE is terminal-based.
- NE access control.

- NE response logging.

In addition, the Virtual Screen NEP provides some action functions which perform NE specific functions such as lookups from static tables and NE-specific data formatting.

Table 5-31 Core NEP Action Functions

Action Function	Description	Java Method	Notes
ERROR	Deprecated.	-	VirtualScreen class.
RESPONSELOG VS_STOP_RESP VS_COPY_RESP VS_SEND_RESP	Start, stop logging of NE responses.	startResponselog() stopResponselog() copyResponselog() returnResponselog()	TelnetConnection class.
VS_GET_RESP VS_SEND_RESP SCREEN_RESP	Create switch history from the response log.	loadResponselog() returnResponselog() returnVirtualScreen()	TelnetConnection class.
SEND	Sends data.	send()	TelnetConnection class.
SENDKEY	Sends a key to a network element.	sendKey()	TelnetConnection class.
MSGSEND	Sends data.	write	SocketConnection class. Send binary data.
MSGRECV	Receives data.	read	SocketConnection class. Receive binary data.
SETOPTION	-	setOption()	TelnetConnection class.
DMS_LEN DMS_FEATS DMS_NAME GET_INCPT GET_LTG GET_P_PARMS GET_SW_FEAT	Deprecated. Applicable only to DMS NEs.	-	-

ADD_HEADER

Adds a specified header into the message that is ready to be sent to an EDD, and is awaiting a header.

See also "MSGSEND."

Syntax:

```
ADD_HEADER '%RTN=%VAR:TYPE'
```

Parameters:

- **%RTN:** The return variable (either SUCCEED or FAIL).
- **%VAR:** The variable that includes a message. A header type that must be one of the following:
  - CONNECT

- CONNECTED
- CONNECT_FAIL
- DISCONNECT
- DISCONNECTED
- DATA
- NO_ACTION
- BREAK_KEY
- OPTION
- DEBUG
- DEBUG_DISCONN

Example:

```
1000 BCONCAT   '%MESSAGE=%MSG1:%MSG2:%MSG3'
1010 ADD_HEADER   '%RTN=%MESSAGE:DATA'
1020 MEGSEND   '%MESSAGE'
1030 MSGRECV   '%RTN=MESSAGE_ACK'
```

ASC_TO_BIN

Converts the value of the numeric variable, **%VAR**, from ASCII data type to binary.

The maximum length of the ASCII variable must be:

- **decimal** – 10 characters or less
- **octal** – 11 characters or less
- **hexadecimal** – 8 characters or less

This action function does not issue error messages when there are invalid characters in the string for the specified base.

For octal conversion, within the limit of 2^32-1, the action function does not allow the string to have the 11th character specified because the code only checks for 10 characters regardless of the base.

Syntax:

```
ASC_TO_BIN '%RTN=%VAR:TYPE'
```

Parameters:

Table 5-32 ASC_TO_BIN parameters

Name	Description	Req'd	Input/Output
%RTN	The return variables are: • SUCCEED • FAIL	4	O
%VAR	The variable to be used by the action function.	4	I

Table 5-32 (Cont.) ASC_TO_BIN parameters

Name	Description	Req'd	Input/Output
TYPE	Data type. Must be one of the following: • O – OCTAL_FIELD • D – DECIMAL_FIELD • H – HEX_FIELD The default value is D.	8	I

Example:

```
100  BCONCAT   '%MSG1=32'
110  BCONCAT   '%MSG2=37'
120  CONCAT   '%AsciiVar=49'
130  ASC_TO_BIN   '%RTN=%AsciiVar:D'
140  BCONCAT   '%MESSAGE=%MSG1:%MSG2:%AsciiVar'
150  MSGSEND   '%MESSAGE'
```

ASC_TO_BIN will cause %AsciiVar to have 31 which the binary equivalent of 49.

ASDL_EXIT

Used in State Table programs that access NEs to complete an ASDL command. The programs then return the appropriate error, classified by an error type, and an optional error description. ASDL exit types include:

- SUCCEED – Successful ASDL command execution.
- FAIL – Hard error.
- RETRY – An ASDL command failed, but retries.
- MAINTENANCE – An ASDL command failed because the NE is currently unavailable to receive provisioning requests.
- SOFT_FAIL – An ASDL command failed but processing continues; does not result in failure of the order.
- DELAYED_FAIL – An ASDL had failed during provisioning. The SARM skips any subsequent ASDL in the CSDL, continues provisioning at the next CSDL, and then fails the order.
- STOP – Stops the ASDL command from processing.

Refer to the *ASAP Cartridge Development Guide*for more detailed descriptions of these base_types.

Java equivalent – setASDLExitType. Sets the exit type and returns from the function.

Syntax:

```
ASDL_EXIT '%ERR_TYPE:%ERR_DESC'
```

Parameters:

- **%ERR_TYPE:** A required parameter. Specifies the error type for the ASDL command. This can be one of the base error types or a user-defined type created using the Enhanced Error Management functionality.

- **%ERR_DESC:** An optional parameter. Optional error description field that can be used to provide a descriptive message about the error that occurred.

Example 1:

```
1100  IF_THEN     '%ANALOG == "N"'
1110    CONCAT     '%OPTION=CDB'
1120    CHAIN     'M-SET_OPTION_ON'
1130  ENDIF     ''
1150  CONCAT     '%ERR_TYPE=U_FAIL1'
1160  CONCAT     '%ERR_DESC=Unable to add Feature 1001'
1170  ASDL_EXIT     '%ERR_TYPE:%ERR_DESC'
```

Example 2:

```
|
|
2000  CONCAT       '%msg=Fail adding 3 features for :%MOBILE_NUMBER:'
2010  CONCAT       '%msg=%msg: - Mobile number is invalid'
2020  LOG       '%msg'
2030  VS_SEND_RESP        ''
2040  ASDL_EXIT       'INVALID_NUMBER:%msg'
```

BIN_TO_ASC

Converts the value of the binary variable, **%VAR**, from binary to a numeric ASCII value.

The maximum length of the ASCII variable must be:

- **decimal** – 10 characters or less
- **octal** – 11 characters or less
- **hexadecimal** – 8 characters or less

This action function does not issue error messages when there are invalid characters in the string for the specified base.

For octal conversion, within the limit of 2^32-1, the action function does not allow the string to have the 11th character specified because the code only checks for 10 characters regardless of the base.

See also ASC_TO_BIN, MSGRECV.

Syntax:

```
BIN_TO_ASC '%RTN=%VAR:TYPE'
```

Parameters:

Table 5-33 BIN_TO_ASC parameters

Name	Description	Req'd	Input/Output
%RTN	The return variables are • SUCCEED • FAIL	4	O
%VAR	The variable to be used by action function.	4	I

Table 5-33 (Cont.) BIN_TO_ASC parameters

Name	Description	Req'd	Input/Output
TYPE	Data type. Must be one of the following. • O – OCTAL_FIELD • D – DECIMAL_FIELD • H – HEX_FIELD The default value is D.	8	I

Example:

```
1000  ASC_TO_BIN  '%RTN=%AsciiVariable:D'
1010  CONCAT  '%MESSAGE=%MSG1:VALUE=
       :%AsciiVariable:%MSG2'
1020  MSGSEND  '%MESSAGE'
1030  MSGRECV  '%RTN=MESSAGE_ACK:%Length'
1040  NVIS_PARSE  '%MESSAGE'
1050  BIN_TO_ASC  '%RTN:%AsciiVariable:D'
```

CLEAR_VS

Clears the virtual screen.

See also "GET."

Syntax:

```
CLEAR_VS''
```

ERROR

Deprecated.

Java equivalent – VirtualScreen class.

GET

Retrieves data from the virtual screen at a specified location. If the data retrieved from the virtual screen does not match the specified value, it retries once every second for num_retries times.

ROW, COL, and LEN are required keywords. ROW and COL specify the screen location using (1,1) as the top left corner. The lower right corner of the screen is specified by the appropriate communication parameters in the device configuration, for instance 80, 24.

Relative screen row positioning is specified using C for the current row only, for example, C-1 in row implies the line above the current row.

It is also possible to use state table variables for ROW, COL, and LEN and wildcard matches are possible for ROW and COL. The wildcard character is “*".

If the **GET** action references areas outside of the virtual screen, it returns to the calling state table. Check the value returned by the **GET** action in the state tables.

Syntax:

```
GET'ROW:row:COL:col:LEN:len:INT:retry_interval %var=value:' num_retries
OR
GET'ROW:row:COL:col:LEN:len:%var=value:' num_retries
```

Parameters:

- **ROW:row:** The row number. Possible values are:
  - A number – For example, **ROW:4:**.
  - A relative location – For example, **ROW:C-2:** – two rows above the current one.
  - A variable – For example, **ROW:%Row:** – references the variable specified in %Row.
  - A wildcard – For example, for any row on the screen, specify **ROW:*:**. If wildcards are used, the search value must be specified.
- **COL:col:** The column number. Possible values are:
  - A number – For example, **COL:4:**.
  - A variable – For example, **COL:%Col:** – references the variable specified in %Col.
  - A wildcard – For example, for any column on the screen, specify **COL:*:**. If wildcards are used, the search value must be specified.
- **LEN:len:** The length of the field to be retrieved. Possible values are:
  - A number – For example, **LEN:4:**.
  - A variable – For example, **LEN:%Len:** –references the variable specified in %Len.
- **%var:** Specifies the NEP variable that stores the value read from the virtual screen. This variable is referenced in the State Table after the GET call.
- **value:** Specifies the value expected to be read. This value can be hard-coded or passed through a variable.

  If a value is not specified, the virtual screen is read and no checks are performed.

  The GET action retries for **num_retries** times in this case. If the value is not being specified, you must specify **num_retries** as **1** (the default).

  If wildcards are used as either the row or column specifiers, you must specify this value.
- **INT:retry_interval:** Optional function that specifies the time in seconds to wait between attempts.
- **num_retries:** The maximum number of retries. If a number is not specified, it defaults to **1**.

For the state table to determine the coordinates at which a match was found (for instance, using wildcard matching), the GET action defines two reserved State Table variables. These variables specify the matching **x** and **y** coordinates. These are only created by the **GET** action in non-loopback mod e. Possible values are:

- **VS_X_COORD:** The column offset at which point a match was found.
- **VS_Y_COORD:** The row offset at which point a match was found.

Example:

```
# Search for "Journal File" at row %Row, col %Col and len %Len
1000   GET  'ROW:%Row:COL:%Col:LEN:%Len:%Var=Journal
     File' 5
# Search for "Journal File" anywhere on the current line
1010  GET    'ROW:C:COL:*:LEN:12:%Var=Journal File'  5
# Search for "Journal File" anywhere on the virtual screen
1020  GET    'ROW:*:COL:*:LEN:12:%Var=Journal File'  5
# Search for "Journal File" at row 10, col 1 and len 12
1030  GET     'ROW:10:COL:1:LEN:12:%Var=Journal File'  5
# Search for "IBM test user"
2000  CONCAT     '%Row=3'
2010  CONCAT     '%Len=13'
2020  GET     'ROW:%Row:COL:*:LEN:%Len:%prompt=IBM test user' 10
# Only reference the returned coordinates if not in Loopback Mode
2030  IF_THEN   '%LOOPBACK_ON == 0'
2040   DIAG   'SANE:[%prompt] Found @ (%VS_X_COORD, %VS_Y_COORD)'
2050  ENDIF   ''
```

GET_INCPT

Deprecated.

GET_LTG

Deprecated.

GET_P_PARMS

Deprecated.

GET_SECUREDATA

Retrieves a user-defined secure data entry.

See also "SET_SECUREDATA."

Syntax:

```
GET_SECUREDATA   '%RTN=%NAME'
```

Arguments:

- **%RTN:** The return value of the action function.
- **%NAME:** The name of the secure data entry used as a key to retrieve the encrypted data.

GET_SW_FEAT

Deprecated.

LOG

Generates a miscellaneous message and transmits it as part of the NE history information to the SARM to be stored in the SARM database.

When logging a compound variable, the variable must appear in braces (as indicated in the example). Otherwise, only the first part of the variable is parsed.

Java equivalent – log method, JProcessor class. Generates an NE history log item.

Syntax:

```
LOG '<expand string>'
```

Example:

```
2030  CONCAT '%MSG=msg'
2330  LOG 'Output Text = [%{MSG.msg_txt}]'
```

LOG_STAT

Deprecated.

MSGSEND

Sends a binary message to the NE when the communication with the NE is message-based. The binary message must be built as per ASAP-NE protocol and saved in **%BVAR** prior to invoking this action.

See also "ADD_HEADER," "ASC_TO_BIN," "MSGRECV."

Java equivalent – write method. SocketConnection class.

Syntax:

```
MSGSEND '%RTN=%BVAR'
```

Example:

```
# Build a specific record and send it
1005 BUILD_MSG  '%LENGTH=%REC_TYPE:PACKET'
1010 IF_THEN  '%LENGTH < 1'
1020 CONCAT  '%ERR_MSG=BUILD_MSG Action Error'
1025 CALL  'SPACE_LIB::ERR_EXIT'
1030 ENDIF  ''
1040 MSGSEND  '%RTN=%PACKET'
1050 IF_THEN  '%RTN != "SUCCEED"'
1051 CONCAT  '%ERR_MSG=MSGSEND Failed'
1052 CALL  'SPACE_LIB::ERR_EXIT'
1070 ENDIF  ''
```

MSGRECV

Receives binary messages from the NE when the communication with the NE is message-based. The binary message of length LEN is expected to be received from the NE and saved in **%BVAR** by this action. The length LEN can be specified using a hard-coded value or a variable can also be passed.

See also "BIN_TO_ASC," "MSGSEND."

Java equivalent – read method. SocketConnection class.

Syntax:

```
MSGRECV '%RTN=%BVAR:LEN' wait_time
```

Example:

```
# Recv Hdr msg for Request-Completion notification
# NOTE: The value of LEN is hardcoded to 12
1190  MSGRECV     '%RTN=PACKET:12'
1200  IF_THEN     '%RTN != "SUCCEED"'
1201  CONCAT     '%ERR_MSG=MSGRECV Failed for HDR_MSG'
1210  CALL     'SPACE_LIB::ERR_EXIT'
1220  ENDIF     ''
1230  EXTRACT_MSG     '%RTN=HDR_MSG:PACKET'
1240  IF_THEN     '%RTN != "SUCCEED"'
1241  CONCAT     '%ERR_MSG=EXTRACT_MSG Failed for HDR_MSG'
1250  CALL     'SPACE_LIB::ERR_EXIT'
1260  ENDIF     ''
# Recv CMD_RESP Msg and extract it
# NOTE: The value of LEN is saved in the variable %RET_PAYLOAD_LENGTH
1270  MSGRECV     '%RTN=PACKET:%RET_PAYLOAD_LENGTH'
1280  IF_THEN     '%RTN != "SUCCEED"'
1281  CONCAT     '%ERR_MSG=MSGRECV Failed for CMD_RESP_MSG'
1290  CALL     'SPACE_LIB::ERR_EXIT'
1300  ENDIF     ''
```

Parameters:

- **%RTN: SUCCEED** – If the action function was successful.
- **%BVAR:** Stores binary data.
- **LEN:** Amount of data to retrieve.
- **wait_time:** Number of seconds to wait. **<0** = infinite wait

NVIS_PARSER

Deprecated.

PARAM_GROUP

Sets the parameter group field to be passed back on any information parameters generated within the State Table. The information parameters that are saved to the SARM database are associated with the specified parameter group in the database.

When querying the SARM for any generated information parameters, the SRP specifies this parameter group as part of the query criteria.

For instance, if you want to return multiple instances of the same data from the NEP State Tables to the SRP (such as configuration details on each of a number of trunk lines), then prior to calling **SEND_CMPND**, you must specify the parameter group for this instance using **PARAM_GROUP**. The SRP then queries on the information parameters using both the parameter group as well as the parameter labels.

See also "SEND_PARAM," "SEND_COMPND."

Java equivalent – setParamGroup method. JProcessor class. Sets a group identifier.

Syntax:

```
PARAM_GROUP '<expand string>'
```

Example:

```
1000  PARAM_GROUP     'GROUP1'
1010  SEND_PARAM     'PARM1 "TEST1" I'
```

```
1020  SEND_PARAM    'PARM1-A "TEST1A" I'
1030  ...
1040  PARAM_GROUP  'GROUP2'
1050  SEND_PARAM  'PARM2 "TEST2" I'
1060  SEND_PARAM  'PARM2-A "TEST2A" I'
```

RESPONSELOG

Enables the response log for any data that is returned from the NE. It causes the Interpreter to open the NE History file for an SRQ. The file can be transmitted back to the SARM using the **VS_SEND_RESP** action.

Upon completion of an ASDL, the Interpreter removes this file regardless of whether its contents were transmitted back to the SARM or not.

See also "VS_COPY_RESP," "VS_GET_RESP," "VS_SEND_RESP," "VS_STOP_RESP."

Java equivalent – startResponselog method. TelnetConnection class.

Syntax:

```
RESPONSELOG ''
```

Example:

```
1000 RESPONSELOG ''
2000 VS_SEND_RESP
```

SCREEN_RESP

Sends the data represented in the virtual screen coordinates that you specify to the SARM as an NE response.

See also "RESPONSELOG," "VS_COPY_RESP," "VS_SEND_RESP," "VS_STOP_RESP."

Java equivalent – returnVirtualScreen method. TelnetConnection class.

Syntax:

```
SCREEN_RESP'X1:Y1:X2:Y2'
```

SEND

This action string is expanded and then sent to the NE when the communication with the NE is terminal-based.

See also "SENDKEY."

Java equivalent – send method. TelnetConnection class.

Syntax:

```
SEND  'string'
```

Example:

```
# Send the NEW command to the NE with parameters $, NXX and LINE
1000 SEND 'NEW $ %NXX%LINE'
1010 SENDKEY 'ENT'  # Send the "ENTER KEY" to apply the MML.
```

SEND_COMPND

Transmits the specified CSDL, Global, Rollback or Information Compound parameters with the specified base name back to the SARM.

See also "SEND_PARAM," "PARAM_GROUP."

Java equivalent – returnCSDLParam, returnCompoundCSDLParam, returnGlobalParam, returnCompoundGlobalParam, returnInfoParam, returnCompoundInfoParam, returnRollbackParam, returnCompoundRollbackParam. JProcessor class. Returns parameters to SARM.

Syntax:

```
SEND_COMPND 'basename flags, NOT_NULL'
SEND_COMPND 'basename flags'
```

Parameters:

- **basename:** Identifies the basename for the Compound parameter.
- **flags:** A string that has one or more of the following values:
  - **C** – CSDL Parameter
  - **G** – Global SRQ Parameter
  - **R** – Rollback Log Parameter
  - **I** – Work Order Information Parameter
- **NOT_NULL:** Only variables that are NOT_NULL are sent to the SARM.

Example:

```
#
# Send Scalar parameters from LEN_HDR structure
#
1000  SEND_PARAM  'LEN_HDR.LEN "%LEN" RI'
1010  SEND_PARAM  'LEN_HDR.MCLI "%MCLI" RI'
#
# Send the entire LEN_HDR structure as a compound structure
# Send Return and Information Parameters
#
1020  SEND_COMPND  'LEN_HDR RI'
```

SENDECHO

Deprecated.

SENDKEY

Sends a function key to an NE. Upon receiving a response, it delays for the sleep value (in seconds), that you specify. This action is applicable only if the communication with the NE is terminal-based.

See also "SEND."

sleep_value is an optional parameter; when no sleep_value is specified, a value of zero (seconds) is passed. The function key, **BRK**, ignores the **sleep_value** option after receiving a response from the switch. Otherwise, the **sleep_value** option is not ignored.

If the sleep value time interval is set to "-1", the action function will return without waiting for a response.

The following values are valid function keys.

Table 5-34 Function key values

Value	Meaning
“ENT"	Enter key
“ESC"	Escape key
“BRK"	Sends the break condition to the NE
“SET"	Sends no data to NE, waits for data
“F01 - F24"	Function keys F1 to F24
“PF1 - PF4"	Function key PF1 to PF4
“AET"	Application Keypad Enter key
“A00 - A09"	Application Keypad 0 to 9
“DOT"	Application Keypad “."
“CMA"	Application Keypad “,"
“DSH"	Application Keypad “-"
ALT-F1 through ALT-F12	Function keys F1 to F12 (Alt)
CTRL-F1 through CTRL-F12	Function keys F1 to F12 (CRTL)
Shift-F1 through Shift-F12	Function keys F1 to F12 (Shift)
CTRL_A through CTRL_Z	Sends control characters (A through Z)

Java equivalent – sendKey method. TelnetConnection class.

Syntax:

```
SENDKEY 'function_key' sleep_value
```

Example:

```
#Send the MML to the switch, followed by the enter key to apply the command.
1000 SEND 'NEW $ %NXX%LINE'
1010 SENDKEY 'ENT' 1
```

SETOPTION

Performs option negotiation with the peer end (NE) where the communications is client-server/peer based. This action is primarily used by the TELNET Interface to set telnet options. The names of the telnet options include:

Table 5-35 Telnet options

Telnet Option	Description
TELOPT_BINARY	8-bit data path
TELOPT_ECHO	echo
TELOPT_RCP	prepare to reconnect

Table 5-35 (Cont.) Telnet options

Telnet Option	Description
TELOPT_SGA	suppress to go ahead
TELOPT_NAMS	approximate message size
TELOPT_STATUS	give status
TELOPT_TM	timing mark
TELOPT_RCTE	remote controlled transmission and echo
TELOPT_NAOL	negotiate about output line width
TELOPT_NAOP	negotiate about output page size
TELOPT_NAOCRD	negotiate about CR disposition
TELOPT_NAOHTS	negotiate about horizontal tab stops
TELOPT_NAOHTD	negotiate about horizontal tab disposition
TELOPT_NAOFFD	negotiate about formfeed disposition
TELOPT_NAOVTS	negotiate about vertical tab stops
TELOPT_NAOVTD	negotiate about vertical tab disposition
TELOPT_NAOLFD	negotiate about output LF disposition
TELOPT_XASCII	extended ascii character set
TELOPT_LOGOUT	force logout
TELOPT_BM	byte macro
TELOPT_DET	data entry terminal
TELOPT_SUPDUP	supdup protocol
TELOPT_SUPDUPOUTPUT	supdup output
TELOPT_SNDLOC	send location
TELOPT_TTYPE	terminal type
TELOPT_EOR	end of record
TELOPT_TUID	TACACS user identification
TELOPT_OUTMRK	output marking
TELOPT_TTYLOC	terminal location number
TELOPT_3270REGIME	3270 regime
TELOPT_X3PAD	X.3 PAD
TELOPT_NAWS	window size
TELOPT_TSPEED	terminal speed
TELOPT_LFLOW	remote flow control
TELOPT_LINEMODE	linemode option

Usage:

- **1** – to set option (ON)
- **0** – to unset Telnet option (OFF)

The following is the function that sets Telnet options:

```
CS_RETCODE Telnet_set_option(CMD_PROC_DATA *data, CS_CHAR *telnet_option,
CS_CHAR *value)
```

The following is an example of case TELOPT_TTYPE:

```
   case TELOPT_TTYPE:
          if (option_value) {
              /* The scenario will be:
               *      . send "will TELOPT_TTYPE"
               *      . recv "do TELOPT_TTYPE"
               *      . recv "send TELOPT_TTYPE"
               *      . send "TELOPT_TTYPE is XXXX"
               */
     .
     .
     .

   else {
              /* The scenario will be:
               *      . send "wont TELOPT_TTYPE"
               *      . recv "dont TELOPT_TTYPE"
               */
```

Telnet _set_option() converts value to integer. You cannot place plain text in the value argument of SETOPTION.

For more information on TELNET options, refer to the UNIX man pages.

The following State Table program provides an example of the TELOPT_ECHO option name to set the telnet session to stop echoing.

```
#%VALUE is 0 or 1
1005 CONCAT '%TE=TELOPT_ECHO'
1006 CONCAT '%OFF=0'
1011 SETOPTION '%SO=%TE:%OFF'
```

Java equivalent – setOption method. TelnetConnection class.

Syntax:

```
SETOPTION '%RTN=%LABEL:%VALUE'
```

SEND_PARAM

Sends the specified variable as a network element parameter to the SARM.

See also "SEND_COMPND," "PARAM_GROUP."

Java equivalent – returnCSDLParam, returnCompoundCSDLParam, returnGlobalParam, returnCompoundGlobalParam, returnInfoParam, returnCompoundInfoParam, returnRollbackParam, returnCompoundRollbackParam. JProcessor class. Returns parameters to SARM.

Syntax:

```
SEND_PARAM 'label value flags'
```

Parameters:

- **label:** The parameter name (Max. 80 characters).

- **value:** The variable or quoted string specifying the parameter. Use the escape character ('\') when quotation characters (**“ “**) are used in a string value.
- **flags:** A string that has one of the following values:
  - **C** – CSDL Parameter (adds parameter to current CSDL).
  - **G** – Global SRQ Parameter (adds parameter to work order).
  - **R** – Rollback Log Parameter (reverses ASDL).
  - **I** – Work Order Information Parameter (sends data back to the SRP).

For more information, see **SRP_asap_get_wo_param** in the SRP Library of the ASAP API Reference).

Example:

```
#
# Send Scalar Parameters
#
# Send CSDL parameter
1000  SEND_PARAM  'CFN "%CFNDN" C'
# Send a global parameter
1010  SEND_PARAM   'MDN_DEFINED "YES" G'
# Send information parameter
1020  SEND_PARAM  'LEN_HDR.MCLI "%MCLI" I'
```

SEND_RESP

Deprecated. Use "VS_SEND_RESP."

SET_SECUREDATA

Updates or adds a user-defined secure data entry.

See also "GET_SECUREDATA."

Syntax:

```
SET_SECUREDATA '%RTN=%NAME:%VALUE'
```

Arguments:

- **%RTN:** Return value of the action function.
- **%NAME:** Data containing information that can be used as a key in the secure data storage.
- **%VALUE:** The data to be secured.

STATS_ON

Deprecated.

VS_COPY_RESP

Stops the logging of information from the NE and then copies the response log file to the filename that you specify. The pathname is preceded with the current **$LOGDIR**, for example, **$LOGDIR/pathname**.

See also "RESPONSELOG," "VS_GET_RESP," "VS_SEND_RESP," "VS_STOP_RESP."

Java equivalent – copyResponselog method. TelnetConnection class.

Syntax:

```
VS_COPY_RESP'pathname'
```

Example:

```
# Copy Switch history to sub-directory keyed by the SITE
# code using LEN number as the filename.
1000 VS_COPY_RESP 'LEN%SITE/%LEN'
...
```

VS_GET_RESP

Copies the current response file to the NE history file. The specified pathname is preceded with the current **$LOGDIR**, for example, **$LOGDIR/pathname**.

See also "RESPONSELOG," "VS_COPY_RESP," "VS_SEND_RESP," "VS_STOP_RESP."

Java equivalent – loadResponselog method. TelnetConnection class.

Syntax:

```
VS_GET_RESP'pathname'
```

Example:

```
# Copy data from the specified file to the switch history file.
2000  VS_GET_RESP  'LEN%SITE/%LEN'
```

VS_SEND_RESP

Stops the logging of information from an NE and then sends the information to the SARM as switch history.

See also "RESPONSELOG," "VS_COPY_RESP," "VS_SEND_RESP," "VS_STOP_RESP."

Java equivalent – returnResponselog method. TelnetConnection class.

Syntax:

```
VS_SEND_RESP  ''
```

Example:

```
1000  RESPONSELOG   ''
2000  VS_SEND_RESP  ''
```

VS_STOP_RESP

Similar to the **VS_SEND_RESP** action, this action stops the logging of information from the NE. However, **VS_STOP_RESP** does not send the information to the SARM as switch history, but rather as a closed response file that can be parsed/processed by the state table or any chained state table.

See also "RESPONSELOG," "VS_COPY_RESP," "VS_SEND_RESP," "VS_GET_RESP."

Java equivalent – stopResponselog method. TelnetConnection class.

Example:

```
1000 RESPONSELOG    ''
...
2000 VS_STOP_RESP   ''
```

LAM action functions

The LAM (Lexical Analysis Machine) is a parser used for retrieving information from a data file that is being processed. High-level action functions are provided to interface with the LAM and control its operation.

NE response parsing is the process of analyzing and creating data from the standard responses provided by an NE. The LAM is a powerful tool that enables the State Table programmer to vary the State Table execution based on the NE responses.

Every operation that the State Table initiates updates a set of global registers within the Interpreter. The State Table can access these registers through a predefined set of parameter names. Based on the contents of the registers, the State Table can take the appropriate actions. Only LAM actions can be used to update these registers. All other actions use the registers in a read-only manner.

Table 5-36 LAM Action Functions

Action Function	Description	Java Method	Notes
SET_MARK	Create a mark.	-	Supported only in State Tables.
GOTO_MARK	Move to a mark.	-	Supported only in State Tables.
DEF_COLUMN	Define a table column.	-	Supported only in State Tables.
RESET_FILE	Reset file pointer.	-	Supported only in State Tables.
SKIP_ITEMS	Skip forward items.	-	Supported only in State Tables.
SKIP_LINES	Skip forward lines.	-	Supported only in State Tables.
UNDO_READ	Move back from the last read.	-	Supported only in State Tables.
READ_TO_EOL	Read lines.	-	Supported only in State Tables.
READ_FIXED, READ_GROUP, READ_ITEM, READ_LAST	Read fields.	-	Supported only in State Tables.
READ_ROW	Read a row of table data.	-	Supported only in State Tables.
READ_STRING	Read strings.	-	Supported only in State Tables.

DEF_COLUMN

Defines a tabular column field to be read at the specified offset and length. Up to 10 columns can be read at a time, as **col #** is used to identify the data register.

See also READ_ROW."

Syntax:

DEF_COLUMN ‘offset:length' col #

Parameters:

- **offset:** The offset at which the column starts. Starts at position 1. This parameter can be a variable.
- **length:** The length from **offset** at which the column ends. This parameter can be a variable.
- **col #:** The number of the data register for up to 10 columns (0-9).

Example:

```
900  CONCAT     '%START=6'
950  CONCAT     '%LEN=10'
1000  DEF_COLUMN     '1:10'  0
1010  DEF_COLUMN     '11:5'  1
1020  DEF_COLUMN     '%START:%LEN'  2
```

This example defines three columns:

1. Starts at position 1 and has a length of 10.
2. Starts at position 11 and has a length of 5.
3. Starts at position 6 and has a length of 10.

GOTO_MARK

Goes to a mark that has been created in the file.

See also "SET_MARK."

Syntax:

```
GOTO_MARK ' ' mark #
```

Parameters:

- **Mark #:** The number of the mark to GOTO. A maximum of 10 marks (0-9) can be used.

Example:

```
1000  SET_MARK  ''  0
1010  READ_ITEM  ''
1020  Process Item
1030  GOTO_MARK  ''  0
1040  READ_TO_EOL  ''  3
1050  Reprocess using the whole line
```

READ_FIXED

From the current cursor position, this action reads a fixed-length field of sizes (bytes) or to the end of the line.

See also "READ_GROUP," "READ_ITEM," "READ_LAST," "READ_ROW," "READ_STRING," "READ_TO_EOL."

Syntax:

```
READ_FIXED ' ' size
```

READ_GROUP

Reads a group of count fields on the current line into the data registers. If the end of the line is reached before the required number of fields is retrieved, the remaining field is set to the NULL string. Valid value for count is 1-10.

See also "READ_FIXED," "READ_ITEM," "READ_LAST," "READ_ROW," "READ_STRING," "READ_TO_EOL."

Syntax:

```
READ_GROUP ' ' count
```

Example:

```
The following is the current line:
CWT 3WC AUL 2962986 LOD 2962987
If this code is executed:
1000    READ_GROUP  '' 4
The data registers would be:
_D0 = CWT, _D1 = 3WC, _D2 = AUL, _D3 = 2962986
```

READ_ITEM

Reads a field from the data file into register **_D0** where the field is delimited by a space, tab, or new line. This state table action also performs an automatic line feed.

See also "READ_FIXED," "READ_LAST," "READ_ROW," "READ_STRING," "READ_TO_EOL."

Syntax:

```
READ_ITEM ' '
```

READ_LAST

Reads to the last field in the current line.

See also "READ_FIXED," "READ_GROUP," "READ_ITEM," "READ_ROW," "READ_STRING," "READ_TO_EOL."

Syntax:

```
READ_LAST ' '
```

READ_ROW

Reads a fixed-format table row, defined by the current column definition, into the data registers. For columns that do not exist in the row, the data register is set to the NULL string (" ").

See also "READ_FIXED," "READ_GROUP," "READ_ITEM," "READ_LAST," "READ_STRING," "READ_TO_EOL."

Syntax:

```
READ_ROW ' ' trim_flag
```

Parameters:

- **trim_flag:** Can be set to indicate whether or not leading blanks, trailing blanks, or both must be stripped. Possible values are:
  - **0** – No trimming
  - **1** – Leading blanks trim
  - **2** – Trailing blanks trim
  - **3** – Leading and trailing blanks trim

READ_STRING

Reads a string of bytes into register _D0 up to one of the specified delimiters. If the end of the line is reached before one of the delimiters, **_D0** is set to the NULL string, ('' '') and the current file position remains unchanged.

See also "READ_FIXED," "READ_GROUP," "READ_ITEM," "READ_ROW," "READ_ROW," "READ_TO_EOL."

Syntax:

```
READ_STRING 'delimiters' trim_flag
```

Parameters:

- **delimiters:** Specifies the delimiter to use in the read operation. Space is specified as **‘ " "'**.
- **trim_flag:** Can be set to indicate whether or not leading blanks, trailing blanks, or both must be stripped. Possible values are listed in the following table.
  - **0** – No trimming
  - **1** – Leading blanks trim
  - **2** – Trailing blanks trim
  - **3** – Leading and trailing blanks trim

READ_TO_EOL

Reads all bytes from the current line position to the end of the line into register _D0.

See also "READ_FIXED," "READ_GROUP," "READ_ITEM," "READ_LAST," "READ_ROW," "READ_STRING."

Syntax:

```
READ_TO_EOL ' ' trim_flag
```

Parameters:

- **trim_flag:** Can be set to indicate whether or not leading blanks, trailing blanks, or both must be stripped. Possible values are:
  - **0** – No trimming
  - **1** – Leading blanks trim
  - **2** – Trailing blanks trim

- **3** – Leading and trailing blanks trim

RESET_FILE

Resets the LAM file pointer to the beginning of the file.

Syntax:

```
RESET_FILE ` '
```

SET_MARK

Marks a location in the file that can be used to move around in the file. You can define a maximum of 10 marks and all marks are initialized to point to the start of the file.

See also "GOTO_MARK."

Parameters:

- **Mark #:** The number of the mark to set (maximum 10). The mark numbers can range from 0 to 9.

Syntax:

```
SET_MARK ` ' mark #
```

Example:

```
1000  SET_MARK ``   0
1010  READ_ITEM ``
1020  Process Item
1030  GOTO_MARK ``   0
1040  READ_TO_EOL ``   3
1050  Reprocess using the whole line
```

SKIP_ITEMS

Moves the current file position to a location that is an **x** number of items ahead of the current location. This action skips the end of the line.

See also "SKIP_LINES."

Syntax:

```
SKIP_ITEMS ` ' count
```

SKIP_LINES

Moves the file pointer to the beginning of a line that is an **x** number of lines later in the file.

See also "SKIP_ITEMS."

A count value of **1** means to move to the next line in the file.

Syntax:

```
SKIP_LINES ` ' count
```

UNDO_READ

Moves the current file pointer to the location in the file before the last read operation.

See also "READ_FIXED," "READ_GROUP," "READ_ITEM," "READ_LAST," "READ_ROW," "READ_STRING," "READ_TO_EOL."

Syntax:

```
UNDO_READ ` '
```

FTP action functions

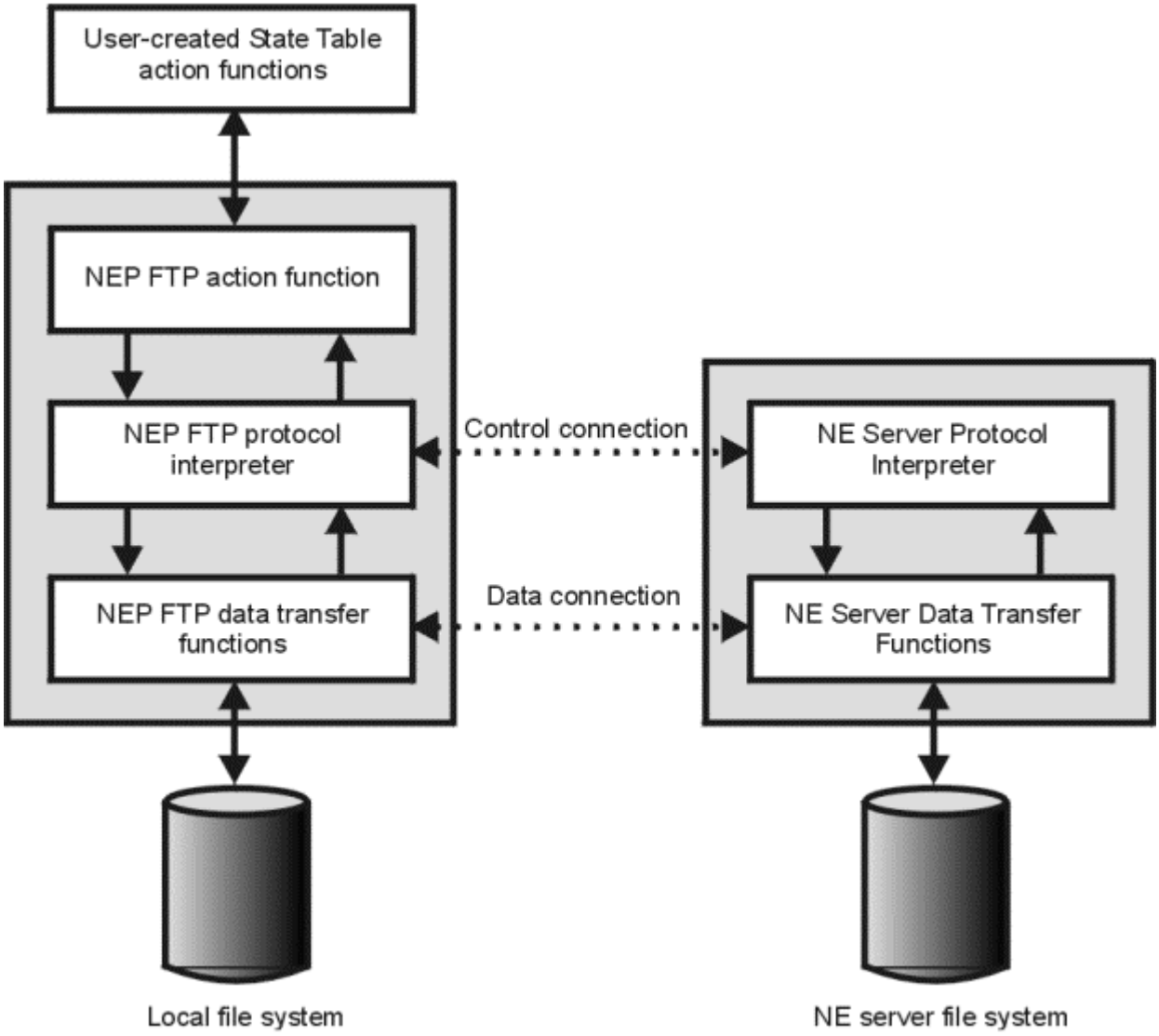
The NEP FTP services are provided as action functions to State Table programs only. Through the user-created State Table programs, you can perform the following FTP functions:

- Receive a file from the NE FTP server.
- Send a file to the NE FTP Server.
- Delete a remote file at the NE file system.
- Change the **File Type** option for transferring files.
- Change the current local working directory.
- Change the current remote working directory.
- Get the name for the current remote working directory.

The Network Element Processor (NEP) is an ASAP application server that executes State Table programs for the communicating operations to the Network Elements in response to ASDL commands received from the SARM. The NEP File Transfer Protocol (FTP) is the FTP client component of the NEP. It provides State Table programs with the ability to delete files, and send and receive files to and from an NE.

Figure 5-4 illustrates the high level architecture of the NEP FTP component communicating with the NE FTP server.

Figure 5-4 NEP FTP Component Architecture



To perform the NEP FTP services, you must use the following State Table action functions:

Table 5-37 State Table Action Functions

Action Function	Description
FTP_APPE	Appends to a file on the NE host server.
FTP_CD	Changes the remote present working directory to the specified path.
FTP_CDUP	Changes the working directory to the parent directory on the NE host server.
FTP_DELE	Deletes a file from the NE host file system.
FTP_DIR	Lists contents of the directory on the NE host server.
FTP_LCD	Changes the local present working directory to the specified path.
FTP_LS	Lists contents of directory on the NE host server.

Table 5-37 (Cont.) State Table Action Functions

Action Function	Description
FTP_MKDIR	Makes a new directory on the NE host server.
FTP_RMDIR	Removes a directory on the NE host server.
FTP_PWD	Gets the remote present working directory.
FTP_RECV	Receives a file from the NE host server.
FTP_REN	Renames a file on the NE host server.
FTP_RUNIQUE	Toggles store unique for local files.
FTP_SEND	Transfers a file to the NE host server.
FTP_SUNIQUE	Toggles store unique on the NE host server.

The general syntax for invoking the action functions is:

```
Line  Action      '%Return=Arguments'
```

Table 5-38 Action Function Syntax

Parameter	Description
Line	The state table program line number.
Action	One of the FTP actions.
Return	A state table variable to store SUCCEED, FAIL, or an FTP-related error code.
Arguments	Parameters that are passed. Arguments can be state table variables.

FTP_APPE

This action causes the local file to be appended to the file on the remote server. If the remote file is not specified, then the remote file is the name of the local file name.

Syntax:

```
FTP_APPE    '%RETURN=LocalFileName:RemoteFileName'
```

Parameters:

- **LocalFileName:** Specifies the local file to append to the file on the remote server. Can be a literal value or a state table variable. Maximum length of the file name is 255 alphanumeric characters.
- **RemoteFileName:** Specifies the remote file to be appended on the remote server. Can be a literal value or a state table variable. Maximum length of the file name is 255 alphanumeric characters.

Return Values:

- **SUCCEED:** The local file is appended to the file on the remote server. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

See also FTP_SEND.

Example 1:

```
1010 FTP_APPE      '%RET=aaa:bbb'
```

Example 2:

```
1010 FTP_APPE      '%RET=/tmp/aaa:bbb'
```

Example 3:

```
1010 FTP_APPE      '%RET=aaa:/tmp/bbb'
```

Example 4:

```
1010 FTP_APPE      '%RET=/tmp/aaa:/tmp/bbb'
```

Example 5:

```
1010  CONCAT        '%LOC_FILE=aaa'
1020  CONCAT        '%REM_FILE=/tmp/bbb'
1030  FTP_APPE      '%RET=%LOC_FILE:%REM_FILE'
```

Example 6:

```
1010  FTP_APPE      '%RET=aaa'
```

FTP_CD

This action causes the remote present working directory to be changed to the directory specified by the argument, directoryName. The directory name can be a full or partial path name.

See also "FTP_LCD" and "FTP_PWD."

Syntax:

```
FTP_CD  '%RETURN=directoryName'
```

Parameters:

- **directoryName:** Specifies the remote host directory to change to. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** The current remote working directory is changed to the specified directory and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP error code.

Example – Changing the current remote working directory:

```
1010  FTP_CD     '%RET=/hpdev/hpenv5/NE_RESPONSES'
1020  FTP_CD     '%RET=LOGS/980315'
```

The current remote working directory after line 1020 is:

```
 /hpdev/hpenv5/NE_RESPONSES/LOGS/980315.
```

FTP_CDUP

This action causes the remote server working directory to be changed to the parent of the current remote server working directory.

See also "FTP_CD" and "FTP_LCD."

Syntax:

```
FTP_CDUP        '%RETURN'
```

Parameters:

None.

Return Values:

- **SUCCEED:** The remote server working directory is changed to the parent of the current remote server working directory. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example:

```
1010  FTP_CDUP      '%RET'
```

FTP_DELE

This action causes the file specified by fileName to be deleted from the remote NE host file system. fileName cannot contain a directory path name. The FTP_CD action is used first to change a remote current directory to the directory containing the file. This action fails if an error occurs, for example, the file does not exist.

See also "FTP_CD."

Syntax:

```
FTP_DELE  '%RETURN=fileName'
```

Parameters:

- **fileName:** Specifies the file name to be retrieved from the remote server. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** Specified file is deleted from the remote server and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or FTP-related error code.

Example 1 – Deleting a file located in the remote present working directory:

```
1010  CONCAT     '%FILE_NAME=NEResp.dat'
1020  FTP_DELE   '%FILE_NAME'
```

The **NEResp.dat** file located in a remote present working directory is deleted.

Example 2 – Deleting a file located in a non-current remote working directory:

```
1010  FTP_CD     '/hpdev/hpenv10/NE_FILES/NE_RESPONSES'
1030  FTP_DELE   'NEResp.dat'
```

The NEResp.dat file is deleted.

FTP_DIR

This action causes a listing of the directory contents of the directory, RemoteDir, on the remote server to be put into the local file, LocalFileName. If the local file name is not specified, then a default local file name, ftp_dir.out, in the current local working directory is chosen.

See also "FTP_LS," "FTP_CD," and "FTP_LCD."

Syntax:

```
FTP_DIR  '%RETURN=RemoteDir:LocalFileName'
```

Parameters:

- **RemoteDir:** Specifies the directory on the remote server to be listed. Can be a literal value or a state table variable.
- **LocalFileName:** Specifies the local file to store the listing of the contents of the directory on the remote server. Can be a literal value or a state table variable. Maximum length of the file name is 255 alphanumeric characters.

Return Values:

- **SUCCEED:** Listing of the directory on the remote server is stored in the local file. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1:

```
1010  FTP_DIR      '%RET=aaa*:bbb'
```

Example 2:

```
1010  FTP_DIR      '%RET=/tmp:bbb'
```

Example 3:

```
1010  FTP_DIR      '%RET=/tmp:/tmp/bbb'
```

Example 4:

```
1010  FTP_DIR      '%RET=-al:/tmp/bbb'
```

Example 5:

```
1010  FTP_DIR      '%RET=-alR:bbb'
```

Example 6:

```
1010  CONCAT       '%REM_DIR=/tmp'
1020  CONCAT       '%LOC_FILE=bbb'
1030  FTP_DIR      '%RET=%REM_DIR:%LOC_FILE'
```

Example 7:

```
1010  FTP_DIR      '%RET=-alR'
```

The listing is stored in the ftp_dir.out file in current local directory.

FTP_LCD

This action causes the local present working directory to be changed to the directory specified by the argument, directoryName. The directory name can be a full or partial path name. This action fails if an error occurs for example, directory does not exist.

See also "FTP_CD" and "FTP_PWD."

Syntax:

```
FTP_LCD  '%RETURN=directoryName'
```

Parameters:

- **directoryName:** Specifies the local host directory to change to. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** The current local working directory is changed to the specified directory and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example – Changing the local present working directory:

```
1010  FTP_LCD  '%RET=/hpdev/hpenv10/NE_RESPONSES'
1020  FTP_LCD  '%RET=/LOGS/980315'
```

The local present working directory after line 1020 is now:

```
/hpdev/hpenv10/NE_REPSONSES/LOGS/980315.
```

FTP_LS

This action results in a listing of the directory contents of the directory, RemoteDir, on the remote server to be put into the local file, LocalFileName. If the local file name is not specified, then a default local file name, ftp_ls.out, in the current local working directory is chosen.

The listing is stored in the ftp_ls.out file in the current local directory.

See also "FTP_DIR," "FTP_CD," and "FTP_LCD."

Syntax:

- **FTP_LS:** '%RETURN=RemoteDir:LocalFileName'

Parameters:

- **RemoteDir:** Specifies the directory on the remote server to be listed. Can be a literal value or a state table variable.
- **LocalFileName:** Specifies the local file to store the listing of the contents of the directory on the remote server. Can be a literal value or a state table variable. Maximum length of the file name is 255 alphanumeric characters.

Return Values:

- **SUCCEED:** The listing of the directory on the remote server is stored in the local file. %RETURN contains SUCCEED.

- **FAIL:** Action fails. %RETURN contains FAIL or an FTP- error code.

Example 1:

```
1010  FTP_LS      '%RET=aaa*:bbb'
```

Example 2:

```
1010  FTP_LS      '%RET=/tmp:bbb'
```

Example 3:

```
1010  FTP_LS      '%RET=/tmp:/tmp/bbb'
```

Example 4:

```
1010  FTP_LS      '%RET=-al:/tmp/bbb'
```

Example 5:

```
1010  FTP_LS      '%RET=-alR:bbb'
```

Example 6:

```
1010  CONCAT       '%REM_DIR=/tmp'
1020  CONCAT       '%LOC_FILE=bbb'
1030  FTP_LS       '%RET=%REM_DIR:%LOC_FILE'
```

Example 7:

```
1010  FTP_LS       '%RET=-alR'
```

FTP_MKDIR

This action creates a new directory on the remote server. The directory name can be a full or partial path name.

Syntax:

```
FTP_MKDIR  '%RETURN=directoryName'
```

Parameters:

- **directoryName:** Specifies the remote host directory to create. It can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** Specified directory is created. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1:

```
1010  FTP_CD     '%RET=/tmp'
1020  FTP_MKDIR  '%RET=sub_tmp'
```

Example 2:

```
1010  CONCAT     '%REM_DIR=sub_tmp'
1020  FTP_CD     '%RET=/tmp'
1030  FTP_MKDIR  '%RET=%REM_DIR'
```

Example 3:

```
1010  CONCAT     '%REM_DIR=/tmp/sub_tmp'
1020  FTP_MKDIR  '%RET=%REM_DIR'
```

FTP_PWD

This action causes the remote server to return to the Present Working Directory. If the action is successful, the state table variable, DirectoryName, is assigned the returned value. This action fails if an error occurs.

See also "FTP_CD" and "FTP_LCD."

Syntax:

```
FTP_PWD  '%RET=%DirectoryName'
```

Parameters:

- **DirectoryName:** An output parameter that contains the name of the remote present working directory. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** Remote present working directory name is returned and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example – Changing the remote present working directory:

```
1010  FTP_CD   '%RET=/hpdev/hpenv5/NE_RESPONSES'
1020  FTP_PWD   '%RET=%DIR'
```

The variable DIR contains the value:

```
 '/hpdev/hpenv5/NE_RESPONSES'
```

FTP_RECV

This action causes the file specified by fileName to be retrieved from the remote NE host server. fileName cannot contain directory path name. The FTP_CD action can be used first to change the remote current directory to the directory where the file is present.

See also "FTP_SEND," "FTP_CD," and "FTP_LCD."

Syntax:

```
FTP_RECV  '%RETURN=fileName'
```

Parameters:

- **fileName:** Specifies the file name to be retrieved from the remote server. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** The specified file is retrieved from the remote server and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP error code.

Example 1 – Retrieving a file and storing it in a non-current local directory:

```
1010  RESPONSELOG    ''
1020  FTP_LCD        '%RET=/hpdev/hpenv5/NE_RESPONSES'
1030  CONCAT         '%FILE_NAME=NEResp.dat'
1040  FTP_RECV       '%RET=%FILE_NAME'
```

The NEResp.dat file is retrieved and stored in the NE_RESPONSES directory. The call to RESPONSELOG initiates the logging of FTP commands and the FTP server replies to be logged on as switch history. You can check the SARM table tbl_srq_log.

Example 2 – Retrieving a file located in a non-current remote working directory :

```
1010  FTP_CD     '%RET=NE_FILES/NE_RESPONSES'
1030  FTP_RECV   '%RET=NEResp.dat'
```

The NEResp.dat file is retrieved and stored in the local present working directory.

FTP_REN

This action causes the file named *from* on the remote server to be renamed as *to*.

See also "FTP_CD" and "FTP_CDUP."

Syntax:

```
FTP_REN  '%RETURN=from:to'
```

Parameters:

- **from:** Specifies the old file name on the remote server. Can be a literal value or a state table variable.
- **to:** Specifies the new file name on the remote server. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** The file on the remote server is renamed to the new file name. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or related FTP error code.

Example 1:

```
1010  FTP_REN      '%RET=aaa:bbb'
```

Example 2:

```
1010  CONCAT       '%FROM=aaa'
1020  CONCAT       '%TO=bbb'
1030  FTP_REN      '%RET=%FROM:%TO'
```

Example 3:

```
1010  CONCAT       '%FROM=aaa'
1020  FTP_REN      '%RET=%FROM:bbb'
```

Example 4:

```
1010  CONCAT       '%TO=bbb'
1020  FTP_REN      '%RET=aaa:%TO'
```

FTP_RMDIR

This action removes a directory on the remote server. The directory name can be a full or partial path name.

See also "FTP_MKDIR," "FTP_CD," "FTP_CDUP," and "FTP_PWD."

Syntax:

```
FTP_RMDIR      '%RETURN=directoryName'
```

Parameters:

- **directoryName:** Specifies the remote host directory to remove. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** Specified directory is removed. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1:

```
1010  CONCAT        '%REM_DIR=/tmp/subtmp'
1020  FTP_RMDIR     '%RET=%REM_DIR'
```

Example 2:

```
1010  CONCAT     '%REM_DIR=sub_tmp'
1020  FTP_CD     '%RET=/tmp'
1030  FTP_RMDIR  '%RET=%REM_DIR'
```

FTP_RUNIQUE

This action function toggles storing of files on the local system with unique file names. If a file already exists with a name equal to the target local file name for FTP_RECV command, a .1 is appended to the name. If the resulting name matches another existing file, a .2 is appended to the original name. If this process exceeds .99, an error message is printed, and the transfer does not take place. The default value is off.

See also "FTP_RECV," "FTP_LS," and "FTP_DIR."

Syntax:

```
FTP_RUNIQUE       '%RETURN=RUNIQUE'
```

Parameters:

- **RUNIQUE:** Can be 0 or 1. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** For the file transfer operations to be successful, local file names are unique. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1:

```
1020  FTP_RUNIQUE  '%RET=0'
```

Example 2:

```
1010  CONCAT  '%RUNIQUE=0'
1020  FTP_RUNIQUE  '%RET=%RUNIQUE'
```

Example 3:

```
1010  CONCAT  '%UNIQ=1'
1020  FTP_RUNIQUE  '%RET=%UNIQ'
```

FTP_SEND

This action causes the file specified by fileName to be sent to the remote NE host server. fileName cannot contain the directory path name. FTP_LCD can be used to change the local directory to where the file is currently located. This action fails if an error occurs, for example, if the file does not exist.

See also "FTP_RECV," "FTP_CD," "FTP_LCD," and "FTP_SUNIQUE."

Syntax:

```
FTP_SEND  '%RETURN=fileName'
```

Parameters:

- **fileName:** Specifies the file name to be transferred to the remote server. Can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** The specified file is transferred to the remote server and the action succeeds. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1 – Transferring a file located in the present local directory:

```
1010  CONCAT     '%FILE_NAME=NEScript.dat'
1020  FTP_SEND   '%RET=%FILE_NAME'
```

The NEScript.dat file is sent and stored in the remote present working directory.

Example 2 – Transferring a file not present in the current working directory:

```
1010  FTP_LCD     '%RET=/hpdev/hpenv10/FTP'
1020  FTP_SEND    '%RET=NEData'
```

The NEData file is sent and stored in the remote present working directory.

FTP_SUNIQUE

This action function toggles the storing of files on the remote system with unique file names. The remote FTP server must support the STOU command for successful completion. The remote FTP server will report the unique name. If a file already exists with a name equal to the target local file name for FTP_SEND command, a .1 is appended to the name. If the resulting name matches another existing file, a .2 is appended to the original name. If this process exceeds .99, an error message is printed, and the transfer does not take place. The default value is Off.

See also "FTP_SEND."

Syntax:

```
FTP_SUNIQUE     '%RETURN=SUNIQUE'
```

Parameters:

- **SUNIQUE:** Can be 0 or 1. SUNIQUE can be a literal value or a state table variable.

Return Values:

- **SUCCEED:** For the file transfer send operations to be successful, remote file names are unique. %RETURN contains SUCCEED.
- **FAIL:** Action fails. %RETURN contains FAIL or an FTP-related error code.

Example 1:

```
1020  FTP_SUNIQUE  '%RET=0'
```

Example 2:

```
1010  CONCAT  '%SUNIQUE=0'
1020  FTP_SUNIQUE  '%RET=%SUNIQUE'
```

Example 3:

```
1010  CONCAT  '%UNIQ=1'
1020  FTP_SUNIQUE  '%RET=%UNIQ'
```

I/O Action Functions

With the following Input/Output (I/O) action functions, you can open, close, read, write, and delete files from State Tables:

Table 5-39 I/O Action Functions

Action Function	Description	Java Method	Notes
OPEN_FILE	Opens the file.	-	Supported only in State Tables.
READ_FILE	Reads from a file.	-	Supported only in State Tables.
WRITE_FILE	Writes ASCII or binary variables to a file	-	Supported only in State Tables.
CLOSE_FILE	Closes a file.	-	Supported only in State Tables.
DEL_FILE	Deletes a file.	-	Supported only in State Tables.

OPEN_FILE

Opens the file whose pathname is stored in the NAME variable. It returns a file handler in the FH variable. You can set the default directory by setting the variable **FILE_DEFAULT_DIR** in the **ASAP.cfg** file. If the variable is not defined in the **ASAP.cfg** file, the present directory is the default directory. For example, if you set the first argument of the action function to **file_name**, then the **file_name** file is searched in "**./**" directory.

You are responsible for opening the file in a proper mode. You can get unpredictable results by opening a binary file in ASCII mode or opening an ASCII file in a binary mode.

Syntax:

```
OPEN_FILE '%FH= %NAME:%MODE'
```

Parameters:

- **FH:** Contains the file handle of the file whose name is stored in NAME variable.
- **NAME:** Variable contains the absolute full path name of the file.
- **MODE:** Contains the file open mode. The following is a list of file modes:
  - **r** – Opens the ASCII file with read-only permission.
  - **rb** – Opens the binary file with read-only permission.
  - **w** – Truncates the ASCII file to zero length or create for writing.
  - **wb** – Truncates the binary file to zero length or create for writing.
  - **a+** – Opens or creates an ASCII file for update at end-of-file. You can also read the file in this mode.
  - **ab+** – Opens or creates a binary file for update at end-of-file.

Return Values:

- **SUCCESS:** A file handler is returned in the FH variable.
- **FAILURE:** The ASDL fails.

READ_FILE

Reads from a file that is represented by the file handler.

ASCII File:

To use the action function, **READ_FILE**, you must open the file in ASCII mode. The action function reads lines or characters from an ASCII file to ASCII variables. The first argument contains the file handler. The second argument represents the variable name that holds characters.

The function returns the number of bytes read from the file. You have the option to place **ALL** in the third argument. The **READ_FILE** then attempts to read the total file. If you place a positive number as the third argument, then the **READ_FILE** reads the number of bytes from the file indicated by the number. If the data length is larger than 254 bytes, it is split to a set of 254 byte long strings. These strings are stored in a set of variables, RBUF_1, RBUF_2, RBUF_3, … RBUF_n.

RBUF is the name of the base variable obtained from the second argument of the action string. While reading a file, the empty lines (only one new line character) is ignored by the action function. If reading only one line, the action ignores multiple empty lines. The action keeps reading the file until it reads a line with characters or encounters the end of the file.

Binary File:

To use the action function, **READ_FILE**, you must open the file in binary mode. The action function reads bytes from a binary file to binary variables. The first argument contains the file handler. The second argument represents the variable name that holds the data.

The function returns the number of bytes read from the file. RBUF is the name of the base variable obtained from the second argument of the action string.

For more information, see the OPEN_FILE action description.

Syntax:

```
READ_FILE '%RTN=%FH:%RBUF:[%OPTION]'
```

Parameters:

- **RTN:** Contains the number of bytes read from the file. If RTN = 0, the end of file is reached.
- **FH:** Contains the file handler that is obtained with **OPEN_FILE** action function.
- **RBUF:** Contains the name of a variable that stores bytes read from the file that has the file handle in the **%FH** variable. The name of the variable can be stored in **%RBUF**. If "**%**" is not prefixed before RBUF, then RBUF is treated as the name of the variable.

OPTION

ASCII File:

This is an optional parameter. The parameter can have positive numbers or a string, **ALL**. If a positive number is defined through the argument, the number represents the number of bytes to be read from the file. If more than 254 bytes are read, the action function splits the data to multiple segments of 254 bytes and stores them in ASCII variables. However, if the read from the file is not larger than 254 bytes, the data is stored in one ASCII variable. For multiple variables, all (except the last variable) contain a string of 254 bytes long. If the parameter is not defined, then all characters of a line (excluding the new line character) are read to a variable. In case a line is more than 254 bytes, the characters read from the file are broken into 254-byte segments.

RTN contains the total number of bytes read. If **ALL** is defined in the OPTION variable, the file is read and stored in multiple ASCII variables. RTN returns the number of bytes read.

Binary File:

This is an optional parameter. The parameter can have positive numbers or a string, **ALL**. If a positive number is defined through the argument, the number represents the number of bytes to be read from the file. RTN returns a positive number to show number of bytes read. It returns 0 to indicate the end of a file. If you set **ALL** as an option, the file is read in the default mode (without the option argument). The default option is to store the file in one variable.

Return Values:

- **SUCCESS:** The number of bytes read is returned in the RTN variable.The RTN variable **0** means you have reached the end of the file for an ASCII file.
- **FAILURE:** The ASDL fails.

Example 1 (ASCII File):

```
READ_FILE  '%RTN =  %FH:RBUF'
```

READ_FILE reads bytes (up to a new line character) from the file (pointed to by file handler, %FH) and stores the data in the variable "RBUF". If the data length is larger

than 254 bytes, but less than (2 x 254 + 1) = 509 bytes, then two variables contain the data.

The first part of the data is stored in **RBUF_0** and the rest is stored in **RBUF_1**. If the data length is less than or equal to 254 bytes, the data is stored in the **RBUF_0** variable. RTN returns the total number of bytes read.

Example 2 (ASCII File):

```
READ_FILE  '%RTN =  %FH:%RBUF'
```

If **FILE_NAME** is stored in the variable **%RBUF**, the **READ_FILE** reads bytes from the file (pointed by the file handle in %FH) until a new line character is encountered and stores the data in variable **FILE_NAME**.

Example 3 (ASCII File):

```
READ_FILE '%RTN = %FH:RBUF:500'
```

The **READ_FILE** reads 500 characters from the file (pointed to by file handler, **%FH**) and stores them in variable(s) that have a base name of **RBUF**. If the file is smaller than 255 characters, the file is stored in the **RBUF_0** variable. If the file is larger than 254 characters but less than 509 characters, then two variables stores the data. The first part of the data is stored in **RBUF_0** and the following is stored in **RBUF_1**. In this case, the RTN returns the total number of bytes read.

Example 4 (ASCII File):

```
READ_FILE '%RTN = %FH:RBUF:ALL'
```

The **READ_FILE** reads the whole file (pointed to by file handler, **%FH**) and stores it over a set of variables that has a base name of **RBUF**. RTN returns the total number of bytes read. If the file is smaller than 255 characters, then the file is stored in one variable. If the file is larger than 254 characters, then multiple variables contain the bytes from the file.

Example 5 (Binary File):

```
READ_FILE '%RTN = %FH:RBUF'
```

The **READ_FILE** reads the total file (pointed to by file handler, **%FH**) and stores it to the variable **RBUF**. RTN returns the number bytes read.

Example 6 (Binary File):

```
READ_FILE '%RTN = %FH:RBUF:100'
```

If the file length is less than or equal to 100 bytes, then **RBUF** contains the whole file and RTN returns the number of bytes read. If the file is larger than 100 bytes, then **RBUF** and RTN contain 100 bytes.

WRITE_FILE

ASCII File:

To use the action function, **WRITE_FILE**, you must open the file in ASCII mode. The action function writes ASCII variables to a file. The first argument is a file handler for the file. The second argument contains the base name of variables that contain strings.

If the second argument is not prefixed with **%**, then the argument is treated as a string and is written to the file. If the third argument is set to **FLUSH**, then the function attempts to flush on write to the file. Writing to the file in such case depends on how the system sets the buffer

default size. If the argument is set to **COMPOUND**, the action function attempts to write all variables to the file. If no option is defined, the action writes one variable specified in the second argument without flushing.

You can place the following control characters

- **\t** – Tab
- **\v** – Vertical tab
- **\r** – Carriage return
- **\n** – New line
- **\f** – Form feed. All other control characters have no effect on the action string.

Binary File:

For more information, see the "OPEN_FILE" action description.

To use the action function, **WRITE_FILE**, you must open the file in binary mode. The action function writes binary variables to a file. The first argument is a file handler for the file. The second argument contains the base name of variables that contain data to be written to the file. If the second argument is not prefixed with **%**, then the action function fails.

If you place **COMPOUND** in the third argument, the action writes all variables that are prefixed with the base name. The **FLUSH** option is similar to the one described for the ASCII file.

> **Note:**
>
> Be careful when defining the size of the buffer for the binary file. If the buffer size defined is 200 and the actual data size is 40, it is the operating system's choice to place any number to occupy the empty space.

Syntax:

```
WRITE_FILE '%RTN=%FH:%WBUF:[%WOPTION]'
```

Parameters:

- **RTN:** Returns the number of bytes written to the file.
- **FH:** Contains the file handle created by the **OPEN_FILE** action.
- **WBUF:** Contains the name of a variable with data that needs to be written to the file. The name can be stored in **%WBUF**. If **%** is not prefixed before WBUF, the WBUF is treated as a string and is written to the file. If you enter **ABC\n** as the second argument instead of **%WBUF**, then **ABC\n** is written to the file. The new line character is **\n**.

WOPTION

This is an optional variable. If you set this argument to **FLUSH**, the action function writes a variable to the file and flushes it. If the argument is set to **COMPOUND**, the function writes contents to the file of all variables in the variable tree that has a base name contained in **%WBUF**. For example, if WOPTION is set to **COMPOUND** and the

base name is WBUF, then the WRITE_FILE action function attempts to read WBUF_1, WBUF_2, … WBUF_n variables from the variable tree and writes them to the file.

Additional variables can be added as long as base name **WBUF** is prefixed in the variable name. For example, WBUF[1], WBUF.1 is written to the file if **COMPOUND** is defined. If the optional variable is not defined, the action writes one variable specified in the **%WBUF** variable.

Return Values:

- **SUCCESS:** The number of bytes written is returned in the RTN variable. If the RTN variable is '0', then no variable is available to write to the file.
- **FAILURE:** The ASDL fails.

The **WRITE_FILE** action follows the Compound parameter behavior. If the base name is **ABC**, then the action function writes contents of all variables that are prefixed by **ABC**. If you have ABCC, ABCD and a set of variables (from the **READ_FILE** action) ABC_1, ABC_2, … ABC_n, then the **WRITE_FILE** writes these variables to a file. If you want to write all variables from the **READ_FILE** action, then the base name must be changed to **ABC_**.

The order of the variables to be written to the file is based on ASCII value of the trailing characters.

CLOSE_FILE

Closes the file related to the file handler contained in **%FH**.

Syntax:

```
CLOSE_FILE  %FH
```

Parameters:

- **FH:** Contains the file handler of the file to be closed.

Return Values:

- **SUCCESS:** Closes the file represented by the file handler.
- **FAILURE:** The ASDL fails.

DEL_FILE

Deletes the file.

Syntax:

```
DEL_FILE  %NAME
```

Parameters:

- **NAME:** Contains the file name that must be deleted.

Return Values:

- **SUCCESS:** Removes the file.
- **FAILURE:** The ASDL fails.

> **Note:**
>
> The file must be closed before the action DEL_FILE is called. You can open the same file twice in a state table, but removing a file from the directory causes a read/write action failure after the removal.

I/O Action Function Error Messages

Message: **WO: YY, Error: File path name is larger than 255 [YY].**

Meaning: Maximum length of the File path is 255 bytes.

Message: **WO: YY, Error: Incorrect number[YY] of arguments.**

Meaning: Check the syntax of the action.

Message: **WO YY, Error: Invalid file open mode [Y] for file [YY].**

Meaning: File open mode is not properly defined in the action string **OPEN_MOD** action. Check the syntax.

Message: **WO YY, Error: Failed to open file YY [AA BB]**.

Meaning: Failed to open the file. Check the unix error message in the diagnostic message. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY, Error: Failed to add the file [YY] in the active file list.**

Meaning: Check the file pointer, file handler and file name, and the file name length. Also, check other messages in th: e diagnostic file.

Message: **WO: YY, Error: Incorrect number [YY] of arguments for READ_FILE.**

Meaning: Check the number of arguments in the action string.

Message: **WO YY, Error: File handler [YY] conversion failed [AA BB].**

Meaning: This points to conversion failure for the file handler. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY, Error:Failed to get the read buffer name for [YY].**

Meaning: Check the second argument of the action string for a proper syntax.

Message: **WO YY, Error: File stat failed [AA BB].**

Meaning: Failed to get file related information. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY, Error: Failed to parse the second argument.**

Meaning: Check the syntax of the second argument of the action string.

Message: **Wrong file type.**

Meaning: Contact Product Support.

Message: **WO YY, Error: Error in the option argument [YY] [AA BB].**

Meaning: Check the third argument of the **READ_FILE** action string for the proper syntax. **AA** represents the unix error message number and **BB** represents the unix error message. The syntax of the action function is important.

Message: **WO YY, Error: Failed to locate the file with the file handler [ZZ].**

Meaning: Check if the file is opened properly. If the problem persists, contact Product Support.

Message: **WO YY, Error: Error in reading the file with the file handler [ZZ] [AA BB].**

Meaning: This is a unix problem. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY, Error: Failed to break the line to multiple lines.**

Meaning: The read data cannot be broken and stored on a multiple variable. Check the data and syntax. Save the diagnostics and contact Product Support.

Message: **WO YY: Error in reading the file pointed by the file handle YY [AA BB].**

Meaning: Check the unix error message. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY, Error: Invalid write argument [YY].**

Meaning: Check the third argument of **WRITE_FILE** for valid options.

Message: **WO YY: Error in writing to the file with the file handler [YY] [AA BB].**

Meaning: Check the unix message. **AA** represents the unix error message number and **BB** represents the unix error message.

Message: **WO YY: Error in getting the file handler value [YY].**

Meaning : Check the action function syntax.

Message: **WO YY: File handler [YY] can not be deleted.**

Meaning: Check the logic of the state table to see if the file was previously opened. If the problem persists, contact Product Support.

Message: **WO YY: Error in getting the file name from [YY].**

Meaning: Check the action string of **DEL_FILE**.

Message: **WO YY: File [YY] deletion failed.**

Meaning: Check the existence of the file in the directory.

Message: **WO YY: File [YY] can not be removed [AA BB].**

Meaning: Check the unix message. **AA** represents the unix error message number and **BB** represents the unix error message.

SNMP action functions

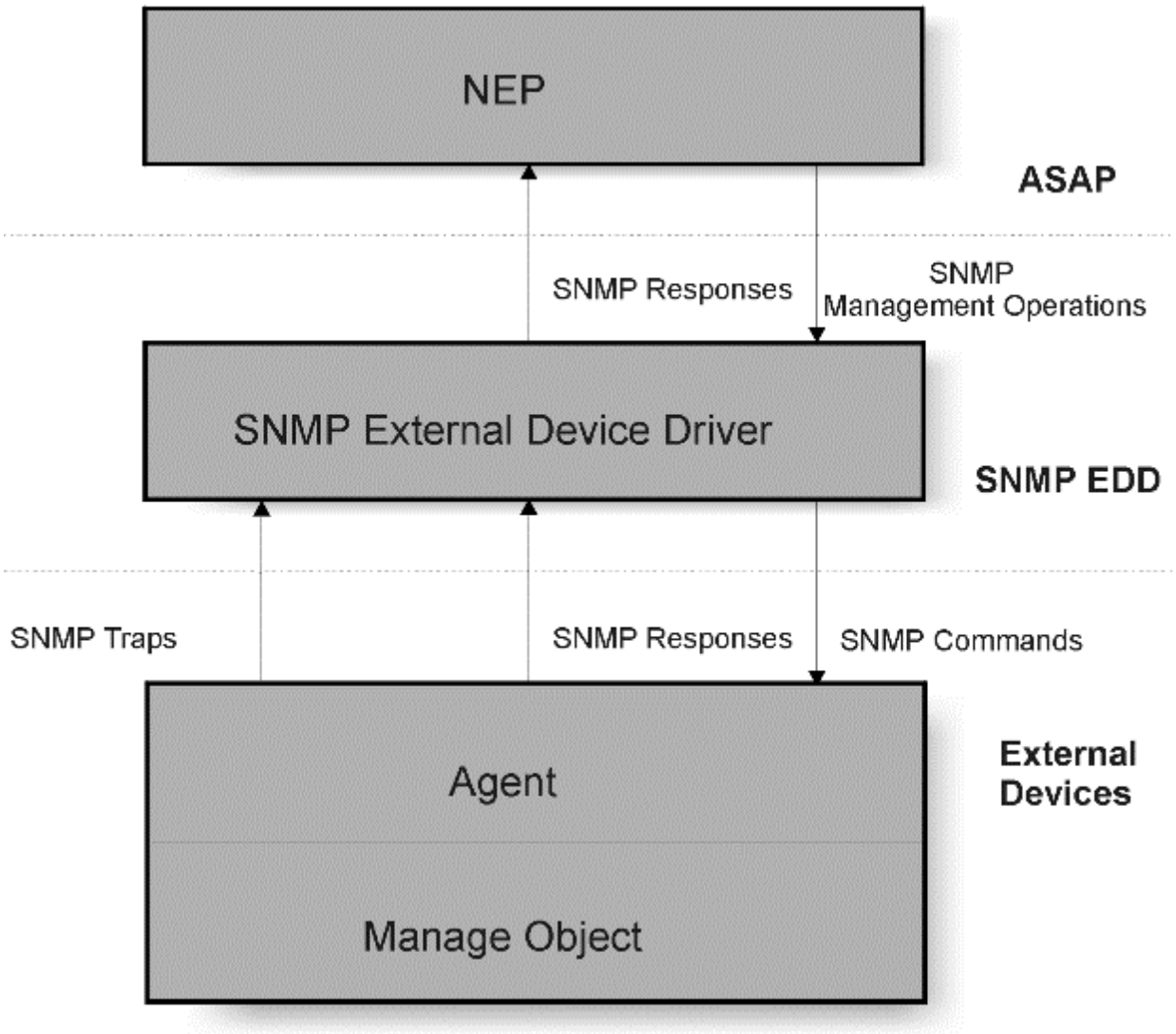
This section details each category of Interpreter State Table action functions. These action functions are supported in State Tables only.

The Network Element Processor (NEP) Simple Network Management Protocol (SNMP) Option provides you with an interface to perform all standard SNMP functions by using the ASAP State Table program to any SNMP network element.

These action functions are supported in State Tables only.

You can write a State Table Program by using the SNMP action functions provided by the NEP. The request is then submitted through the SNMP External Device Driver (EDD) to the network element. The network element sends the correct response back to the NEP through the SNMP EDD and is printed to the ASAP diagnostic file.

Figure 5-5 Communicating with NEs through the SNMP EDD



Variables

Variables in State Table programs are created when you use them for the first time in a State Table. You do not have to define or declare them before they are used for the

first time. If a variable is assigned a value, and does not already exist, it is created. If it does exist, it will be overwritten.

Variable names can consist of the following characters: a to z, A to Z, 0 to 9, ".", "[", "]" and "_", the underscore character.

> **Note:**
>
> The contents of variables is limited to 255 characters. When the length of each action string is greater than 255, an error is generated during compilation of the State Table. The configuration variable STRING_LENGTH_CHECK in ASAP.cfg, permits this from happening. For more information, see STRING_LENGTH_CHECK refer to the *ASAP System Administrator's Guide*.

The curly braces "{" and "}" provide extended variable syntax and are used to inform the Interpreter to explicitly expand their contents first before expanding the rest of the action string. For example, consider the case in which an array index is itself a variable:

```
1000 CONCAT '%TMP=%{ARRAY[%IDX]}'
```

To ensure the correct expansion of this option string, it is essential that **%IDX** be expanded first before the **%ARRAY**. The **%** means that the value has been previously stored and will be substituted by the action function at run-time.

The use of curly braces is highly recommended when fully expanding variable names that include non-alphanumeric characters (especially dot "**.**", left bracket "[", or right bracket "]") carrying special meaning.

Regular expressions

The term **regular expression** is used in the same context as the standard UNIX usage. A formal definition of regular expressions is available in the standard UNIX documentation under regexp. Most UNIX systems have this documentation on-line. It is available by typing the UNIX command:

```
man regexp
```

Expressions:

An expression is any legal combination of symbols that represents a value. An expression is defined as follows:

```
operator::= { ==, !=, <=, >=, <, >, NOT_NULL, IS_NULL, DEFINED, NOT_DEFINED}
expression ::= %var operator [{ %var | number | "string"}]
```

LAM registers

Data registers are a set of global registers in the Interpreter that are updated when the state table initiates an operation. The state table can access these registers by using a predefined set of variable names. Based on these and the contents of the registers, appropriate actions are taken in the state table. Only LAM actions can be used to update these registers. All other actions use the registers in a read-only manner.

The LAM registers are defined below.

Table 5-40 LAM Registers

Register	Description
_D0, .. _D9	Data registers being filled by LAM on read operations. _D0 is the primary data register for single item read operations. The other registers are used when groups and rows are being read.
_EOF	End of file indicator.
_FOFF	Current file offset in bytes from the start of the file.
_FSIZ	Size of the data file in bytes.
_M0,..,_M9	Marking registers used by the LAM to move around in the file.

The SNMP API provides the following actions to construct and submit requests:

Table 5-41 SNMP Action Functions

Action Function	Description	Java Method	Notes
SNMP_GET_REQ	Get Request.	-	Supported only in State Tables.
SNMP_GET_NEXT_REQ	Get Next Request.	-	Supported only in State Tables.
SNMP_GET_BULK_REQ	Get Bulk Request.	-	Supported only in State Tables.
SNMP_SET_REQ	Set Request.	-	Supported only in State Tables.
SNMP_INFORM_REQ	Inform Request.	-	Supported only in State Tables.
SNMP_TABLE_REQ	Table Request.	-	Supported only in State Tables.

The general syntax for invoking the above actions is:

```
Line  Action   '%Ret=Pid:Method:ArgList'
```

SNMP_GET_REQ

With this action you can invoke the SNMP GET request to retrieve the value of one or more variables from an SNMP agent.

Syntax:

```
SNMP_GET_REQ    '%PID=newPid'
SNMP_GET_REQ    '%RET=%PID:setOid:Oid1[…:OidN]'
SNMP_GET_REQ    '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_GET_REQ    '%RET=%PID:setContextName:Name'
SNMP_GET_REQ    '%RET=%PID:setContextID:ContextID'
SNMP_GET_REQ    '%RET=%PID:send'      [Timeout]
```

Table 5-42 SNMP_GET_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new SNMP_GET_REQ. The return identifier is unique across all SNMP requests for the ASDL.

Table 5-42 (Cont.) SNMP_GET_REQ Methods

Method	Description
%RET=%PID:setOid:Oid1[…:OidN]	Adds to the variable binding list of the SNMP request identified by PID object identifiers Oid1, Oid2,…, OidN.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Agent, that is, makes the actual **SNMP_GET_REQ** request invocation. The State Table is blocked until the response is delivered, or a timeout threshold is reached. The timeout threshold is given by the value Timeout (if it is given), or the configured value of the communication parameter **WRITE_TIMEOUT**.

See also "SNMP_RESPONSE," "SNMP_TABLE_REQ."

Table 5-43 SNMP_GET_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.
TIMEOUT	Request timed out.

Example:

```
# Get a fresh PID
1000  SNMP_GET_REQ     '%PID=newPid'
# Get the sysDescr object
2000  SNMP_GET_REQ     '%RET=%PID:setOid:sysDescr.0'
# Submit the request
3000  SNMP_GET_REQ     '%RET=%PID:send'
```

SNMP_GET_NEXT_REQ

This action provides methods to construct an SNMP Get Next request and submit it to the agent. You can invoke the **SNMP_GET_NEXT_REQ** to retrieve the next variable after one or more specified variables from an SNMP agent.

Syntax:

```
SNMP_GET_NEXT_REQ      '%PID=newPid'
SNMP_GET_NEXT_REQ      '%RET=%PID:setOid:Oid1[…:OidN]'
SNMP_GET_NEXT_REQ      '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_GET_NEXT_REQ      '%RET=%PID:setContextName:Name'
SNMP_GET_NEXT_REQ      '%RET=%PID:setContextID:ContextID'
SNMP_GET_NEXT_REQ      '%RET=%PID:send' [Timeout]
```

Table 5-44 SNMP_GET_NEXT_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new SNMP_GET_NEXT_REQ. The return identifier is unique across all SNMP requests for the ASDL.
%RET=%PID:setOid:Oid1[…:OidN]	Adds to the variable binding list of SNMP request identified by PID object identifiers Oid1, Oid2,…, OidN.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Agent, that is, makes the actual SNMP_GET_NEXT_REQ invocation. The State Table is blocked until the response is delivered or a timeout threshold is reached. The timeout threshold is given by the value Timeout (if it is given), or the configured value of the communication parameter WRITE_TIMEOUT.

See also "SNMP_RESPONSE," "SNMP_TABLE_REQ."

Table 5-45 SNMP_GET_NEXT_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.
TIMEOUT	Request timed out.

Example:

```
# Get a fresh PID
1000  SNMP_GET_NEXT_REQ      '%PID=newPid'
# Get the object next to sysUpTime object
2000  SNMP_GET_NEXT_REQ
   '%RET=%PID:setOid:sysUpTime.0'
# Submit the request
3000  SNMP_GET_NEXT_REQ      '%RET=%PID:send'
```

SNMP_GET_BULK_REQ

You can invoke SNMP_GET_BULK_REQ to retrieve large blocks of data efficiently from an SNMP agent.

See also "SNMP_RESPONSE," "SNMP_TABLE_REQ."

Syntax:

```
SNMP_GET_BULK_REQ      '%PID=newPid'
SNMP_GET_BULK_REQ      '%RET=%PID:setOid:Oid1[…:OidN]'
SNMP_GET_BULK_REQ      '%RET=%PID:setNonRepeaters:%Non-Repeaters'
```

```
SNMP_GET_BULK_REQ      '%RET=%PID:setMaxRepetitions:%Max-Repetitions'
SNMP_GET_BULK_REQ      '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_GET_BULK_REQ      '%RET=%PID:setContextName:Name'
SNMP_GET_BULK_REQ      '%RET=%PID:setContextID:ContextID'
SNMP_GET_BULK_REQ      '%RET=%PID:send' [Timeout]
```

Table 5-46 SNMP_GET_BULK_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new SNMP_GET_BULK_REQ. The return identifier is unique across all SNMP requests for the ASDL.
%RET=%PID:setOid:Oid1[…:OidN]	Adds to the variable binding list of the SNMP request identified by PID object identifiers Oid1, Oid2,…, OidN.
%RET=%PID:setNonRepeaters:%Non-Repeaters	Sets the Non-Repeaters parameter of the SNMP_GET_BULK_REQ identified by PID to be %Non-Repeaters.
%RET=%PID:setMaxRepetitions:%Max-Repetitions	Sets the Max-Repetitions parameter of the Get-Bulk request identified by PID to be %Max-Repetitions.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Agent, that is, makes the actual SNMP-Get-Bulk request invocation. The state table is blocked until the response is delivered or a timeout threshold is reached. The timeout threshold is given by the valueTimeout (if it is given), or the configured value of the communication parameter WRITE_TIMEOUT.

Table 5-47 SNMP_GET_BULK_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.
TIMEOUT	Request timed out.

Example:

```
# Get a fresh PID
1000  SNMP_GET_BULK_REQ      '%PID=newPid'
# Set the Non-Repeaters parameter
2000  SNMP_GET_BULK_REQ      '%RET=%PID:setNonRepeaters:1'
# Set the Max-Repetitions parameter
3000  SNMP_GET_BULK_REQ
  '%RET=%PID:setMaxRepetitions:2'
# Get entries in the IP net-to-media table
4000  SNMP_GET_BULK_REQ      '%RET=%PID:setOid:sysUpTime
   :ipNetToMediaPhysAddesss
   :ipNetToMediaType'
```

```
# Submit the request
5000  SNMP_GET_BULK_REQ      '%RET=%PID:send'
```

SNMP_SET_REQ

Invokes the SNMP SET request to set the value of one or more variables from an SNMP agent.

See also "SNMP_RESPONSE," "SNMP_TABLE_REQ."

Syntax:

```
SNMP_SET_REQ    '%PID=newPid'
SNMP_SET_REQ    '%RET=%PID:setOidVal:Oid:Val'
SNMP_SET_REQ    '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_SET_REQ    '%RET=%PID:setContextName:Name'
SNMP_SET_REQ    '%RET=%PID:setContextID:ContextID'
SNMP_SET_REQ    '%RET=%PID:send' [Timeout]
```

Table 5-48 SNMP_SET_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new SNMP Set request. The return identifier is unique across all SNMP requests for the ASDL.
%RET=%PID:setOidVal:Oid:Val	Adds to the variable binding list of SNMP request identified by PID object identifiers Oid and its value Val. A binary value can be set by providing a HEX value with the first 2 characters as 0x prefix. Note that x here is not capital, it has to be a small character.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Agent, that is, makes the actual SNMP-Set request invocation. The State Table is blocked until the response is delivered or a timeout threshold is reached. The timeout threshold is given by the value Timeout (if it is given), or the configured value of the communication parameter WRITE_TIMEOUT.

Table 5-49 SNMP_SET_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.
TIMEOUT	Request timed out.

Example:

```
100 SNMP_SET_REQ '%PID=newPid'
120 SNMP_SET_REQ '%RET=%PID:setOidVal:notifyConfirm.0:0x 07'
```

```
#120 SNMP_SET_REQ '%RET=%PID:setOidVal:notifyConfirm.0:0x 01 03'
#120 SNMP_SET_REQ '%RET=%PID:setOidVal:notifyConfirm.0:0x 0b 1f'
124 SNMP_SET_REQ '%RET=%PID:setAuthInfo:public'
200 SNMP_SET_REQ '%RET=%PID:send'
```

SNMP_INFORM_REQ

Provides methods to construct an SNMP Inform request and submit it to another SNMP entity acting in a manager role.

See also "SNMP_RESPONSE."

Syntax:

```
SNMP_INFORM_REQ     '%PID=newPid'
SNMP_INFORM_REQ     '%RET=%PID:setSysUpTime:Time'
SNMP_INFORM_REQ     '%RET=%PID:setInfoNameVal:Name:Val'
SNMP_INFORM_REQ     '%RET=%PID:setOidVal:Oid:Val'
SNMP_INFORM_REQ     '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_INFORM_REQ     '%RET=%PID:setContextName:Name'
SNMP_INFORM_REQ     '%RET=%PID:setContextID:ContextID'
SNMP_INFORM_REQ     '%RET=%PID:send' [Timeout]
```

Table 5-50 SNMP_INFORM_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new SNMP Inform request. The return identifier is unique across all SNMP requests for the ASDL.
%RET=%PID:setSysUpTime:Time	Sets the value of the first object sysUpTime.0 in the Variable Binding List.
%RET=%PID:setInfoNameVal:Name:Val	Sets the snmpEventID.i as the second object in the Variable Binding List.
%RET=%PID:setOidVal:Oid:Val	Adds to the variable binding list of the SNMP request identified by PID object identifiers Oid and its value Val.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Manager, that is, makes the actual SNMP-Inform request invocation. The State Table is blocked until the response is delivered or a timeout threshold is reached. The timeout threshold is given by the value Timeout. If it is given, the configured value of the communication parameter is WRITE_TIMEOUT.

Table 5-51 SNMP_INFORM_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.

Table 5-51 (Cont.) SNMP_INFORM_REQ Return Values

Return Value	Description
TIMEOUT	Request timed out.

Example:

```
# Get a fresh PID
1000  SNMP_INFORM_REQ     '%PID=newPid'
# Set the sysUpTime and snmpEventID object
2000  SNMP_INFORM_REQ
  '%RET=%PID:setSysUpTime:87956'
3000  SNMP_INFORM_REQ
 '%RET=%PID:setInfoNameVal:surgeBreakerAlarm:0.0'

4000  SNMP_INFORM_REQ
 '%RET=%PID:setOidVal:surgeBreakerStatus.0
    :unknown'
# Submit the request
4000  SNMP_INFORM_REQ     '%RET=%PID:send'
```

SNMP_TABLE_REQ

Provides methods to construct a request to retrieve all the rows in a table and submit it to the agent.

See also "SNMP_RESPONSE."

Syntax:

```
SNMP_TABLE_REQ     '%PID=newPid'
SNMP_TABLE_REQ     '%RET=%PID:setTableName:Table'
SNMP_TABLE_REQ     '%RET=%PID:setAuthInfo:AuthInfo'
SNMP_TABLE_REQ     '%RET=%PID:setContextID:ContextID'
SNMP_TABLE_REQ     '%RET=%PID:setContextName:Name'
SNMP_TABLE_REQ     '%RET=%PID:send' [Timeout]
```

Table 5-52 SNMP_TABLE_REQ Methods

Method	Description
%PID=newPid	Returns an identifier for a new Table request. The return identifier is unique across all SNMP requests for the ASDL.
%RET=%PID:setTableName:Table	Specifies the table name.
%RET=%PID:setAuthInfo:AuthInfo	Sets the authentication information AuthInfo for the SNMP request identified by PID.
%RET=%PID:setContextName:Name	Sets the SNMPv3 Context Name for the SNMP request identified by PID.
%RET=%PID:setContextID:ContextID	Sets the SNMPv3 Context ID for the SNMP request identified by PID.

Table 5-52 (Cont.) SNMP_TABLE_REQ Methods

Method	Description
%RET=%PID:send' [Timeout]	Submits the request identified by PID to the SNMP Agent, that is, makes the actual SNMP-Get-Next or SNMP-Get-Bulk request invocation. The State Table is blocked until the response is delivered or a timeout threshold is reached. The timeout threshold is given by the value Timeout (if it is given), or the configured value of the communication parameter WRITE_TIMEOUT.

Table 5-53 SNMP_TABLE_REQ Return Values

Return Value	Description
SUCCEED	Request completed with no local error.
TIMEOUT	Request timed out.

Example:

```
# Get a fresh PID
1000  SNMP_TABLE_REQ     '%PID=newPid'
# Set the Table Name
2000  SNMP_TABLE_REQ
  '%RET=%PID:setTableName:udpTable'

# Submit the request
4000  SNMP_TABLE_REQ     '%RET=%PID:send'
```

SNMP_RESPONSE

Provides methods to process the SNMP Get-Response for any type of SNMP Request identified by PID.

Syntax:

```
SNMP_RESPONSE '%RSP=%PID:allParams'
SNMP_RESPONSE '%ERR_STAT=%PID:errorStatus'
SNMP_RESPONSE '%ERR_IDX=%PID:errorIndex'
SNMP_RESPONSE '%RET=%PID:hasVarBindList'
SNMP_RESPONSE '%VAR=%PID:varBindList'
```

The methods provided by SNMP_RESPONSE are:

Table 5-54 SNMP_RESPONSE Methods

Method	Description
%RSP=%PID:allParams	Retrieves all parameters of the response for the SNMP request identified by PID. The response is stored into a Compound variable RSP.
%ERR_STAT=%PID:errorStatus	Retrieves the Error Status parameter of the response for the SNMP request identified by PID. The Error Status value is stored into a Scalar variable ERR_STAT.

Table 5-54 (Cont.) SNMP_RESPONSE Methods

Method	Description
%ERR_IDX=%PID:errorIndex	Retrieves the Error Index parameter of the response for the SNMP request identified by PID. The Error Index value is stored into a Scalar variable ERR_IDX.
%RET=%PID:hasVarBindList	Reports whether the Variable Binding List parameter of the response for the SNMP request identified by PID is present. If present, 1 is returned; if not present, 0 is returned.
%VAR=%PID:varBindList	Retrieves the Variable Binding List of the response. If the Variable Binding List is not present, it returns an empty string. Otherwise, for each variable, which is the n-th element of the list, it creates the variable VAR[n].oid and VAR[n].val, and copies to them the object identifier and its value. You must invoke the hasVarBindList method before invoking this method.

Example:

```
# Check whether this is a success or failure response
500  ###### Set the Request ######
1000  '%RET=%PID:send'
1010  IF_THEN   '%RET == "SUCCEED"'
1020   SNMP_RESPONSE  '%ERR=%PID:errorStatus'
1030      IF_THEN  '%ERR != "noError"'
1040       SNMP_RESPONSE  '%IDX=%PID:errorIndex'
1050     ELSE  ''
1060       SNMP_RESPONSE  '%RET=%PID:hasVarBindList'
1070       IF_THEN '%RET == "1"'
1080         SNMP_RESPONSE'%VAR=%PID:varBindList'
1090       ENDIF  ''
1100     ENDIF  ''
1110  ENDIF ''
```

LDAP action functions

Lightweight Directory Access Protocol (LDAP) is an open standard protocol that provides uniform access to Directory Services. The LDAP Action Functions are written so that ASAP implements a LDAP client that is generic and not specific to a given ASAP application. The LDAP action functions provide read and write access to information stored in a directory service at the time of executing state tables in ASAP.

Connectivity to the LDAP Directory Servers (NEs) is provided through the Multi-Protocol Manager. The LDAP state table action functions provide a transparent access to the LDAP directory servers through the LDAP API, a set of functions (or classes) that request the servers to perform operations defined by the LDAP protocol.

ldap Directory Entry Structure

The LDAP directory service is based on a set of entries, a collection of attributes identified by a name called a distinguished name (DN). The DN is a unique entry name. Each of the attributes of an entry has a type and one or more values. The types are mnemonic strings, for example, **cn** for a common name, and the values depend on what type of attribute it is.

The LDAP Action Functions contain the necessary input and output information that is carried in compound variables with syntactical structure. This allows for results of one query to be the input for another. Compound variables in ASAP are constructed by adding a '**.**' followed by a part name to the root name.

For example, the variable **QUERY.DN.ALL:**

- QUERY – Is the root name
- DN – Is a part name
- ALL – Is a sub-part name

Extended State Table variables

State table variables are limited to a maximum size of 255 characters. Since the directory service can store arbitrary data, it is possible that a DN or an attribute value is longer than 255 characters. You can append numbers to compound variable parts and sub-parts to overcome this size limitation. If the attribute value is longer than the 255 characters, the string must be split up into multiple strings (additional buffers) which are assigned to separate compound variable sub-parts by adding '**.**' followed by a number to the part name.

For example, if the string of DN types was 600 characters, you must split it up into three separate strings. The first string is stored in **QUERY._DN**, the second string in **QUERY._DN.0**, and the third in **QUERY._DN.1**. You can apply this structure to any compound variable that is used by any of the LDAP action functions. If the value stored in **QUERY._DN.du0** must be longer than 255 then the second string would go into **QUERY._DN.du0.0**, the third into **QUERY._DN.du0.1** etc.

You must not split the long strings in the middle of a word but on spaces between words. You must also verify that the variables returned from LDAP action functions have been split correctly and you must insert spaces as required. While this approach is cumbersome, in most cases it will not be necessary since the 255 characters size is sufficient.

Keywords:

The special keywords (**_DN**) that are used as parts or sub-parts always start with an underscore to distinguish them from data. There are a limited number of such keywords and they are discussed in the Action Function sections in which they are used.

LDAP Entry Examples:

Most of the LDAP action functions require a Distinguished Name (DN) as one of their inputs. The Distinguished Name is the location of a given entry in a hierarchical directory, for example, “cn=John Smith,, ou=Employee, d=Engineering, c=Canada, o=xyz.com". Since this path can be quite long (over the 225 limit) it is represented with compound variable parts and sub-parts. Therefore, the representation of the example above would be as follows (core name being QUERY):

```
QUERY._DN   = "cn ou d c o"
QUERY._DN.cn   = "John Smith"
QUERY._DN.ou   = "Employee"
QUERY._DN.d   = "Engineering"
QUERY._DN.c   = "Canada"
QUERY._DN.o   = "xyz.com"
```

In the above example, the part **_DN** is set to a string containing all the attribute types in the order in which they appear in a DN. Each attribute type also appears as a sub-part following the **_DN** part and they are set to their respective values. If the same attribute type is repeated

within a DN, a sequence number must be appended to the sub-part. For example, the DN: “cn=John, ou=Employee, du=eng, du=xyz, du=com' would be represented in a following compound variable:

```
QUERY._DN   = "cn ou du du du"
QUERY._DN.cn   = "John Smith"
QUERY._DN.ou   = "Employee"
QUERY._DN.du   = "eng"
QUERY._DN.du0   = "xyz"
QUERY._DN.du1   = "com"
```

The attribute type list appears exactly as in a DN (without any additional numbers) but the sub-parts of recurring attribute types have numbers appended to them starting at 0. Since most of the time a given type only appears once in a DN the first occurrence of **du** type has nothing appended to it. The next **du** type has **0** appended to it, next one has a **1**, and so on.

LDAP Operations:

The main use of LDAP is to search for information in the directory. The LDAP search operation allows a portion of the directory to be searched for entries that match criteria specified by a search filter. A client can request information from each entry that matches the criteria. LDAP defines operations for interrogating and updating the directory. It provides operations for adding and deleting an entry from the directory, changing an existing entry, and changing the name of an entry.

Configuring LDAP:

To configure LDAP, you must do the following:

1. Set the following configuration variable in **ASAP.cfg**:

   LDAP_IF_SUPPORTED=1

2. Set the device type to **W**.

LDAP Configuration Example:

```
var retval number
exec :retval := SSP_new_resource ('$ASAP_SYS',
   'DMS_POOL','W' 1);

exec :retval := SSP_new_comm_param('W','COMMON_HOST_CFG','COMMON_DEVICE_CFG',
 'HOST_NAME','toronto.xyz.com', 'LDAP server host name');
exec :retval := SSP_new_comm_param ('W','COMMON_HOST_CFG','COMMON_DEVICE_CFG',
 'HOST_USERID','cn=manager', 'dc=metasolv', 'userid');
exec :retval := SSP_new_comm_param ('W','COMMON_HOST_CFG','COMMON_DEVICE_CFG',
 'HOST_PASSWORD','secret','user password');
exec :retval := SSP_new_comm_param ('W','COMMON_HOST_CFG','COMMON_DEVICE_CFG',
 'HOST_IPADDR','47.11.145.143', 'IP address);
exec :retval := SSP_new_comm_param ('W','COMMON_HOST_CFG','COMMON_DEVICE_CFG',
 'PORT','390','port number');
```

Communication Parameters

The following communication parameters must be defined:

Table 5-55 LDAP Communication Parameters

Parameter	Description
HOST_NAME	Name of the computer that hosts the directory service (for example, dir.xyz.com) or the IP address of the host.
HOST_USERID	Full DN of an existing user that has appropriate privileges. Maximum 255 characters.
HOST_PASSWORD	User password. Maximum 255 characters.
PORT	Port number of the directory service (optional – default port is 389).
OPEN_TIMEOUT	Duration in seconds to wait for binding to the directory server. Possible values: • Default – 5 seconds • Minimum – 2 seconds • Maximum – 600 seconds
READ_TIMEOUT	Maximum time in seconds to wait for results from the directory server. Possible values: • Default – 1 second • Minimum – 1 second • Maximum – 3600 seconds
LDAP_VERSION	LDAP version to use. If VERSION2, use 2. If VERSION3, use 3. (Default value is 2.)
SIZELIMIT	The size of the search results set asked from the directory server. Possible values: • Default – 2 • Minimum – 1 • Maximum – 500

The LDAP action functions implemented in ASAP are:

- LDAP_SEARCH
- LDAP_COMPARE
- LDAP_ADD
- LDAP_DELETE
- LDAP_MODIFY
- LDAP_RENAME
- STRTOK

LDAP_SEARCH

This function searches the directory for entries that meet specified search criteria.

Syntax:

```
LDAP_SEARCH '%<COUNT>=<SEARCH>:<RESULT>'
```

where:

- <**SEARCH**>: The name of compound variable that holds the search criteria information.
- <**RESULT**>: T he name of compound variable that will hold the search results.

- <**COUNT**>: A return value that indicates the number of result sets. If this action function fails, **FAIL** is returned.

The **Search** compound variable contains the following parts:

Table 5-56 Search compound variable parts

Variable	Description
_DN _DN.<type>	Distinguished name of the directory entry that serves as a starting point (base) for the search. For example, setting this to “o=xyz.com" restricts the search entries at xyz.com.
_SCOPE	Scope of the search. Possible values are: • **BASE** – Searches the entry specified by base. • **ONELEVEL** – Searches all entries one level beneath the entry specified by base. • **SUBTREE** (default) – Searches the entry specified by base and all entries at all levels beneath the entry specified by base.
_FILTER	String representation of the filter to apply in the search. The format for the filter is (attribute operator value) where: **attribute** – Is one of the valid entry attributes as defined by the Directory Server schema. **operator** – Is one of: • **=** – Returns entries whose attribute is equal to the value. • **>=** – Returns entries whose attribute is greater than or equal to the value • **<=** – Returns entries whose attribute is less than or equal to the value • **=*** – Returns entries that have a value set for that attribute • **~=** – Returns entries whose attribute value approximately matches the specified value. Typically, this is an algorithm that matches words that sound alike. **value** – Is the value to match (wildcard * can be used, e.g. “F*" for values starting with F). Search filters can also be combined using the following syntax: `(boolean_operator (filter1) (filter2) (filter3))` where boolean_operator can be one of: • **&** – Returns entries matching all specified filter criteria • **|** – Returns entries matching one or more of the filter criteria • **!** – Returns entries for which the filter is not true. You can only apply this operator to a single filter, for example: (!(filter)).
_ATTR	String of attributes to return from the entry. If this keyword is omitted, all attributes of the entry are returned. Servers do not normally return operational attributes in search results unless you specify the attributes by name.

Search Results:

The result of a search are stored in compound variable(s) that has the name passed in <RESULT> and has a result set number between 0 and <COUNT>-1 appended to it. For example, if the name of the result variable is OUT and the number of results is 4, then the variables OUT0, OUT1, OUT2, and OUT3 are created by the action function.

The **Result Set** variable contains the following parts:

Table 5-57 Result set variable

Variable	Description
_DN _DN.<type>	The Distinguished Name for the found entry formatted as described above.
_DN._ALL	This convenience variable stores the DN, as returned by the server, in one string. Use this variable when you need to send the entire DN to another system. For example, “cn=Arthur, ou=Person, c=Canada".
_ATTR	List of attributes returned by this search, for example, “cn ou c".
<type>	Value for the specific attribute, for example, OUT1.cn, OUT1.ou, OUT1.c.

Example:

For more information on the STRTOK function, see "STRTOK."

The following is an example of a state table that performs a simple search and sends the results back to the switch history using the **LOG** action function. This state table is using a string tokenizer function, **STRTOK**. In addition, there are no checks performed for additional buffers (_DN.0, _ATTR.0, etc.). State table line numbers have been omitted.

```
BEGIN LDAP_SEARCH_EXAMPLE
# Setup search parameters
CONCAT '%IN._DN=o'
CONCAT '%IN._DN.o=xyz.com'
CONCAT '%IN._SCOPE=ONELEVEL'
CONCAT '%IN._FILTER=(cn=a*)'
# Peform search
LDAP_SEARCH '%COUNT=IN:OUT'
# Check for FAIL return value should be done here
# Process search results and store in switch history
CONCAT '%I=0'
CONCAT '%LAST=0'
WHILE '%I < %COUNT'

STRTOK '%TOKEN=%{OUT%I._DN}:" ":LAST'
WHILE '%TOKEN != ""'
CONCAT '%VALUE=%{OUT%I._DN.%TOKEN}'
LOG 'DN%I %TOKEN ->  %VALUE'
STRTOK '%TOKEN=%{OUT%I._DN}:" ":LAST'
ENDWHILE
CONCAT '%LAST=0'
STRTOK '%TOKEN=%{OUT%I._ATTR}:" ":LAST'
WHILE '%TOKEN != ""'
CONCAT '%VALUE=%{OUT%I.%TOKEN}'
LOG 'ATTR%I %TOKEN ->  %VALUE'
STRTOK '%TOKEN=%{OUT%I._ATTR}:" ":LAST'
ENDWHILE
INCREMENT '%I'
CONCAT '%LAST=0'

ENDWHILE
ASDL_EXIT 'SUCCEED'
END LDAP_SEARCH_EXAMPLE
```

LDAP_COMPARE

This function checks if an attribute of an entry contains specified value.

Syntax:

```
LDAP_COMPARE '%<RET>=<DN>:<ATTRIBUTE>:<VALUE>'
```

where:

- <**DN**>: The name of a compound variable that holds the Distinguished Name. The <DN> compound variable requires the **_DN** part.
- <**ATTRIBUTE**>: The name of the attribute.
- <**VALUE**>: The attribute value passed in the parameter that will be compared to its value in the directory.
- <**RET**>: Possible values are:
  - **TRUE** – Match is found
  - **FALSE** – No match is found
  - **FAIL** – Operation failed

Example:

```
CONCAT '%COM._DN=cn ou o'
CONCAT '%COM._DN.cn=John Smith
CONCAT '%COM._DN.ou=People'
CONCAT '%COM._DN.o=xyz.com'
LDAP_COMPARE '%RET=COM:sn:Smith
SEND_PARAM 'COMP_RESULT "%RET" I'
ASDL_EXIT 'SUCCEED'
```

LDAP_ADD

This function inserts a new entry into the directory.

Syntax:

```
LDAP_ADD '%<RET>=<VAR>'
```

where:

- <**VAR**>: A name of a compound variable that holds the DN, attributes and their values to be added
- <**RET**>: A return value that takes SUCCEED and FAIL values.

The **Add** compound variable contains the following parts

Table 5-58 Add compound variable parts

Variable	Description
_DN _DN<type>	Distinguished name of the entry to add. With the exception of the leftmost component, all components of the distinguished name (for example, o=organization or c=country) must already exist.
_ATTR	List of attributes of the entry to add, for example, “cn ouc"

Table 5-58 (Cont.) Add compound variable parts

Variable	Description
<type>	Values for the specified attributes.

Example:

```
CONCAT '%ADD._DN=cn ou o'
CONCAT '%ADD._DN.cn=John Smith"
CONCAT '%ADD._DN.ou=People'
CONCAT '%ADD._DN.o=xyz.com'
CONCAT '%ADD._ATTR=objectclass cn sn'
CONCAT '%ADD.objectclass=top'
CONCAT '%ADD.objectclass.0=person'
CONCAT '%ADD.cn=John Smith'
CONCAT '%ADD.sn=Smith'
LDAP_ADD '%RET=ADD'
ASDL_EXIT 'SUCCEED'
```

LDAP_DELETE

This function deletes an entry from the directory.

Syntax:

```
LDAP_DELETE '%<RET>=<DN>'
```

where:

- <**DN**>: A name of a compound variable that holds the DN of the entry that needs to be deleted.
- <**RET**>: A return value that takes SUCCEED and FAIL values.

The <**DN**> compound variable needs the **_DN** part specified as in other functions.

Example:

```
DIAG 'LOW:LDAP_DELETE Example'
CONCAT '%DEL._DN=cn ou o'
CONCAT '%DEL._DN.cn=John Smith'
CONCAT '%DEL._DN.ou=People'
CONCAT '%DEL._DN.o=xyz.com'
LDAP_DELETE '%RET=DEL'
SEND_PARAM 'DeleteRes "%RET" I'
ASDL_EXIT 'SUCCEED'
```

LDAP_MODIFY

This function modifies an existing directory entry.

Syntax:

```
LDAP_MODIFY '%<RET>=<VAR>'
```

where:

- <**VAR**>: A compound variable name that holds the DN, attributes and their modifications.

- <**RET**>: A return value that takes SUCCEED and FAIL values.

The **Modify** compound variable contains following parts:

Table 5-59 Modify compound variable parts

Variable	Description
_DN _DN.<type>	Distinguished Name of an existing entry that will be modified.
_ATTR	List of attributes that are added, deleted or replaced.
<type>._OP	Indicates type of operation performed for the given attribute type. Possible values are: • ADD – Adds the given attribute and its value to the entry • DEL – Deletes the given attribute and its value from the entry • REP – Replaces the existing attribute value with a new one. • If the <type>._OP is not specified, then the ADD operation is assumed.
<type>	Value for the specified attributes. Not required for the DEL operation.

Example:

```
CONCAT '%MOD._DN=cn ou o'
CONCAT '%MOD._DN.cn=John Smith'
CONCAT '%MOD._DN.ou=People'
CONCAT '%MOD._DN.o=xyz.com'
CONCAT '%MOD._ATTR=sn'
CONCAT '%MOD.sn_OP=REP'
CONCAT '%MOD.sn=Smith66666'
LDAP_MODIFY '%RET=MOD'
SEND_PARAM 'CompareRes "%RET" I'
ASDL_EXIT 'SUCCEED'
```

LDAP_RENAME

This function changes the DN of an entry in the directory. You use this function to change the DN of a given entry (right most part of a DN) or to move a given entry to another location in a directory by changing the parent of a DN (all left side).

Syntax:

```
LDAP_RENAME '%<RET>=<VAR>'
```

where:

- <**VAR**>: A name of a compound variable that holds the DN and other necessary information
- <**RET**>: A return value that takes SUCCEED and FAIL values.

The **Rename** compound variable contains following parts:

Table 5-60 Rename compound variable parts

Variable	Description
_DN _DN.<type>	Distinguished Name of an existing entry that will be renamed or moved.
_NEW._DN _NEW._DN.<type>	New Relative Distinguished Name (RDN, rightmost part) to assign to the entry.
_PAR._DN _PAR._DN.<type>	If the entry is being moved this part specifies the new parent (left side) of the DN. This part can be omitted if entry is not being moved. **Note**: The _PAR._DN.<type> variable is used in LDAP version 3 only.
_DEL	If present and set to TRUE then the old RDN value is deleted. By default, the old RDN value is appended as another value to the given RDN attribute.

Example:

```
CONCAT '%MOD._DN=cn ou o'
CONCAT '%MOD._DN.cn=John Smith'
CONCAT '%MOD._DN.ou=People'
CONCAT '%MOD._DN.o=xyz.com'
CONCAT '%MOD._NEW._DN=cn'
CONCAT '%MOD._NEW._DN.cn=John Smith the Third'
LDAP_RENAME '%RET=MOD'
SEND_PARAM 'RenameRes "%RET" I'
ASDL_EXIT 'SUCCEED'
```

STRTOK

This function takes a string to be tokenized and returns the string that is split up, on a specified delimiter returning one token per call.

Syntax:

```
STRTOK '%<TOKEN>=<STR>:<DELIM>:<LAST>'
```

where:

- <**STR**>: The string to tokenize. The input string <**STR**> should not be changed while it is tokenized.
- <**DELIM**>: A string containing delimiter characters. The value of the <**DELIM**> string can change between calls.
- <**LAST**>: The name of the variable that stores the position of the last found token. Set this value to 0 when starting to tokenize a new string and do not change it between calls to the STRTOK function.
- <**TOKEN**>: The return value that is set to the last found token.

In addition, the input string <STR> should not be changed while it is tokenized. The value of the <DELIM> string can change between calls.

6
Web Services

This chapter describes the Oracle Communications ASAP's Web Services interface. It includes the following topics:

Web Services Overview

ASAP provides a web services interface through which external applications can manage service activation activities and operations. Using the web services interface, you can develop distributed platform-and-application-server agnostic in-house solutions.

The interface is defined in the ASAP Web Services Web Service Definition Language (WSDL) file.

ASAP Web Services runs on Oracle WebLogic Application Server. See the discussion on specific version numbers of mandatory and optional third-party software in *ASAP Installation Guide*.

The external transport protocols are HTTP, HTTPS, and JMS and the data service formats are SOAP v1.1 and 1.2.

Access-level security is provided through the implementation of the WebLogic Server WS-Policy specification, enforcing authentication.

ASAP Web Services takes advantage of the existing JSRP functionality to interact with the ASAP core.

Web Services Definition Language (WSDL)

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of which message formats or network protocols are used in implementation. Type the following URL in your web browser and access the ASAP Web Services WSDL:

`http://server:port/env_id/Oracle/CGBU/Mslv/Asap/Ws?WSDL`

where:

- *server* is the server address.
- *port* is the port number of the Oracle WebLogic Server installation.
- *env_id* is the environment identifier specified when ASAP was installed.

HTTP protocol is used for a handshake with the application server to authenticate and request a web service client stub, which is used as the launch pad to talk to the web service. Then the client can communicate with the ASAP Web Services using one of the HTTP, HTTPS, or JMS protocols.

Architectural Overview of Web Services

ASAP Web Services supports the OSS/J standard to interact with web services clients. The ASAP Web Services WSDL works with values compliant with com.sun.java.product.oss data types. See the TM Forum website at:

http://www.tmforum.org/browse.aspx?catID=2896

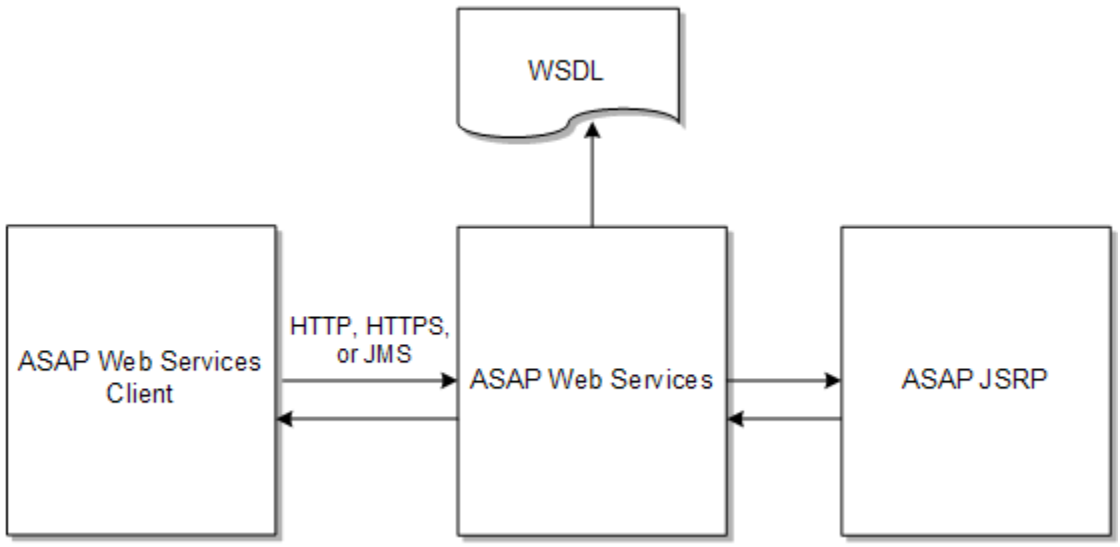
Web services exposes a web interface according to the ASAP Web Service WSDL contract for ASAP Service Activation. Web services clients can take advantage of the web services interface in accordance with ASAP's WSDL contract to exploit service activation functionality. This interface works with OSS/J compliant data types.

The ASAP Web Services message validator will check the incoming messages for compliance with ASAP OSS/J data types (for example, com.sun.java.product.oss.xml.serviceactivation.OrderValue message values actually need to be com.metasolv.serviceactivation.ASAPOrderValue values) and if incoming messages are of the expected types, submits the request to the JSRP. The JSRP forwards the request to SARM, and the results will be returned to the web services client.

Currently, web services does not support asynchronous operations and does not return JSRP event queue messages back to the clients. Web services creates a temporary queue but this queue is only for the purpose of notification of an operation completion.

Figure 6-1 illustrates the flow of messages between ASAP Web Services client and the JSRP.

Figure 6-1 ASAP Web Services with JSRP



Web Services Interface

The ASAP Web Services WSDL exposes most of the functionality that is available through JSRP in ASAP for flow through activation. See "About Web Service Operations" for more information about WSDL operations.

ASAP Web Services must use OSS/J compliant data types. The WSDL file defines message types according to the OSS/J-type XML files with ASAP extensions. You can refer to the following XSD files provided with the ASAP application for extension descriptions:

- XmlCommonSchema.xsd
- XmlServiceActivationSchema.xsd
- ASAPServiceActivation.xsd

See *ASAP Online Reference* for more information about the XSD files. The *ASAP Online Reference* can be extracted from the *ASAP_src***/doc.tar** file, where *ASAP_src* is the location of the ASAP installation files. The annotated XSD files can also be found in the ASAP environment at *ASAP_Home***/xml/xsd**.

The WSDL document declares the OSS/J elements for each web services operation. Following is a snippet for the startOrderByKey operation in a WSDL document:

```
<xs:element name="startOrderByKey">
<xs:complexType>
<xs:sequence>
<xs:element ref="s4:startOrderByKeyRequest"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

For ASAP, we need to pass a raw XML document in OSS/J plus ASAP extension type.

Security

ASAP Web Services access control security determines the functionality that each user will be able to access. In order to set up access control security, create a security role. Give this role the privilege to invoke ASAP Web Services. When the web services client needs to access the web service, the client will need to authenticate itself to the Oracle WebLogic Server hosting ASAP Web Services. (Refer to the WebLogic Server Administration Guide for details on how to set up access security.)

Note: WebLogic Server access control security only protects WebLogic Server resources and does not cover secure communication with ASAP Web Services. As a result, SOAP messages transmitted between the web service and its invoking clients are in plain text.

Currently, web services only offers access level security. Clients must use a user ID that is a member of group **ASAP_WS_USERS_GROUP** to communicate with ASAP WebServices. The **web.xml** file defines the security role **ASAP_WS_USERS** and **weblogic.xml** file defines the security principal name as **ASAP_WS_USERS_GROUP**. The ASAP installer creates a default user named **asap_ws_user**. This user is a member of the **ASA_WS_USERS_GROUP** group. Due to limitations of the WebLogic Administration Console, information created by the command-line tools such as the role name may not be available in the console.

About Web Service Operations

Table 6-1 lists the supported and unsupported web services OSS/J Common Schema base operations.

Table 6-1 Web Services Common Schema Base Operations

Supported	Unsupported
getManagedEntityTypes	getEventDescriptor
getQueryTypes	getEventTypes
getSupportedOptionalOperations	

Table 6-2 lists the supported and unsupported web services OSS/J Service Activation Schema base operations.

Table 6-2 Web Services Service Activation Schema Base Operations

Supported	Unsupported
abortOrderByKey	getOrdersByTemplates
createOrderByValue	getSupportedOptionalAttributes
getOrderByKey	makeOrderValue
getOrdersByKeys	makeServiceValue
getOrderTypes	orderAttributeValueChangeEvent
getServiceTypes	orderCreateEvent
queryOrders	orderRemoveEvent
removeOrderByKey	orderState
resumeOrderByKey	orderStateChangeEvent
setOrderByValue	priority
startOrderByKey	serviceState
suspendOrderByKey	trySetOrdersByValues
tryAbortOrdersByKeys	
tryCreateOrdersByValues	
tryRemoveOrdersByKeys	
tryStartOrdersByKeys	

Table 6-3 lists the supported and unsupported web services OSS/J ASAP Service Activation Schema base operations.

Table 6-3 Web Service ASAP Service ACtivation Schema Base Operations

Supported	Unsupported
cancelOrderByKey	abortService
lockOrder	addExtendedOrderProperty
stopOrderByKey	addOrderParameter
unlockOrder	addServiceParameter
	addService
	deleteService
	getInitOrderByKey
	orderCompleteEvent
	orderEstimateEvent
	orderFailEvent
	orderNEUnknownEvent
	orderRollbackEvent
	orderSoftErrorEvent
	orderStartupEvent
	orderTimeoutEvent
	orderTimeoutWarningEvent
	removeExtendedOrderProperty
	removeOrderParameter
	removeServiceParameter
	resubmitOrderByKey
	retryService
	setExtendedOrderProperty
	setOrderParameter
	setServiceParameter
	validateOrderOperation
	validateServiceOperation

A

Sample Thread Framework Application

EDD framework is a layer on the top of the thread framework. This layer provides basic functionality of the EDD which allows the application to use its API to monitor th e connections to NEP, receive data from the NEP and send data to NEP.

The design of the EDD framework in this section is an example of using thread framework. EDD Data Architecture.

Many other features used by EDD such as logging diagnostic messages, generating events and retrieving configuration variables are described in document -- MT-Safe Common Class Library, which is part of the framework also.

Essentially, the framework maintains one thread for each connection to NEP. A dedicated thread is generated at startup time to listen connecting requests initiated by NEP. These threads are generated from the classes inherited from **ASC_ThreadAppl**. Typically, the EDD application should subclass from these two classes to build the application specific classes. This is to be demonstrated in the section describing DCE API.

The classes used in this framework are:

- EDD Connection Listening
- Connection Handler Class

EDD connection listening class

This **EDD_Listen** class provides the functionality of listening to incoming connecting requests from NEP and generating the connection handler threads to handle connections to NEP. This is very similar to a server spawning a thread to handle a request. The object created from this class is attached to a DCE thread.

This class provides functionality to handle connection request both the UNIX domain socket and Internet domain socket. Once a request is accepted, the listener spawns an **EDD_ConnHandler** thread to handle the connection.

This is an abstract class that you must subclass from. The main reason for subclassing this class is to provide a method to instantiate connection handler objects, since you must subclass **EDD_ConnHandler** to build the application.

The following is the class definition.

Synopsis

```
class EDD_Listener : public ASC_ThreadAppl
{
public:
EDD_Listener(char* Listener);
// This is the entry point which is called when the
// thread is up. It, in turn, calls function to
// monitor the listening socket.
int threadMain(void **Rtn);
```

```
// This function has to be redefined by the
// application to generate connection handler.
// Usually, a new command is used to create a
// new connection handler object.
virtual EDD_ConnHandler *genConnHandler(const

ConnStartInfo &ConnInfo) = 0;

protected:
Diagnosis *m_Diag;
Event *m_Event;
Config m_Config;
private:
// This function is called by threadMain to
// initialize the listening socket.
void initListener(void);
// This function retrieves listening file descriptor.
void getListenFileDes(int SocketFamily, char *IPAddr,

  int Port);

// Constructs listening socket for UNIX domain.
int unixDomainSocket(char *PathName);
// Constructs listening socket for internet domain.
int inetDomainSocket(char *HostIPAddr, short Port);
// This function accepts a request to establish a
// socket connection to NEP.
int newConnection(void);
// The following two connections are used by
// newConnection to establish a socket connection to NEP.
int unixSocketConnect(int SocketID);
int inetSocketConnect(int SocketID);
int m_FileDes;
};
```

Public methods

```
int threadMain(void **Rtn);
```

This function is invoked when the thread is up. It initializes the listening channel and then blocks the thread on UNIX system call **accept()**. The thread is woken up when a request is coming. Upon accepting the request, the thread spawns a connection handler thread to handle it. Most functions in the private section are directly or indirectly used by this function.

```
virtual EDD_ConnHandler *genConnHandler(const ConnStartInfo &ConnInfo) = 0;
```

This function is used to generate a connection handler thread. **ConnInfo** is passed to the new thread object to transfer the file descriptor. It returns the pointer to the new thread. The user has to redefine this function. Refer to the DCE API section for examples.

The following is the flow of the control:

1. **ThreadMain()** calls **initListener()**.
2. **initListener** calls either **unixDomainSocket** or **inetDomainSocket** depending on the configuration.
3. Once the listening channel is established, **newConnection** is called to establish listening channel.

4. **newConnection** invokes either **unixSocketConnect** or **inetSocketConnect** to wait for the requests.
5. Once a connection is established, a connection handler is spawned to handle the connection.

Connection handler class

This is an abstract class that you derive the application from. You must redefine some member functions which are invoked for different transactions (see the Transaction chapter for detail). For example, when the NEP requests to establish a connection to a network element or a remove server, the member function **connectReq** is called. You redefine this function to establish the connection to network elements.

After the object of the class is created, it attaches to a thread. The object then invokes the **readSocket** member function which blocks the thread to wait for data coming from NEP. Based on the header information in data read, the object invokes proper functions redefined by the application to handle requests.

The following describes those member functions.

Synopsis

```
class EDD_ConnHandler : public ASC_ThreadAppl
{
public:
EDD_ConnHandler(char *ConnectionName);
~EDD_ConnHandler(void){}
// This is the entry point of the thread.
int threadMain(void **Rtn);
// This function is called when the thread is just up,
// so that the user may use this function to perform
// their own initializetion.
virtual void applInitialize(void) = 0;
// The user may use the following two functions to
// save the application specific data. setApplData
// calls the user defined deleteApplData internally
// to make sure that there is no memory leak.
void setApplData(void *ApplData);
void *getApplData(void) { return m_ApplData; }
// The user has to redefine this function to delete
// the data that was saved using setApplData().
virtual void deleteApplData(void *ApplData) = 0;
// Redefine this function to clean up the application
// data. This function is called whenever the thread
// is terminated.
virtual void cleanup(void) = 0;
// This function is called whenever NEP requests a
// connection to a remote server or network element.
// The connection to the NEP is closed if this
// function returns ASC_Fail.
virtual int connectReq(void) = 0;
// The user uses this function to retrieve connection
// parameters stored in tbl_comm_param of SARM database.
const char *getConnParam(const char *ParamLabel);
// This function is called whenever NEP requests to send
// data or perform some operation on remote server. The
// application has to redefine this one to provide
```

```
// the service.
virtual int processDataReq(const unsigned char *Data,

  const int Len) = 0;

// This one allows the user to pass data to NEP.
int sendDataToNEP(const unsigned char *Data,

const int Len);

// This function is called whenever NEP requests to
// close a connection to a remote server or network
// element. The connection to the NEP is closed
// whenever this function is returned.
virtual void disconnectReq(void) = 0;
// This function is provided for the user to close the
// connection to NEP whenever it detects that connection
// to remote server is gone.
void disconnectACK(void);
virtual void sendKeyReq(void *data, int len) = 0;
void setFileDes(const int FileDes){ m_FileDes=FileDes; }
const int getFileDes(void) { return m_FileDes; }
// These three objects are used by DCE routines to
// log diagnostic messages and events and also
// get configuration variables.
Diagnosis *m_Diag;
Event *m_Event;
Config m_Config;
protected:
private:
// send ACK and NACK to NEP
void connectACK(void);
void connectNACK(void);
// Retrieve EDD header
int getEddHeader(int &Len, int &Type);
// Build a EDD header to data to be sent to NEP.
int buildEddHeader(unsigned char **Buf, const

unsigned char *data, const int DataLen,
const int DataType);

// Retrive connection data. The data contains
// parameters saved in  tbl_comm_param of SARM dB.
const char* getConnectionData(const int ExpectedLen);
// These two functions read and write to sockets
// connect to NEP.
int readSocket(unsigned char *Buffer, const

int ExpectedLen);

int writeSocket(const unsigned char *Buffer, const

int Length);

// This function configures the socket.
int tuneFileDes(void);
char *m_ConnParam;
int m_DataType;
int m_FileDes;
void *m_ApplData;
};
```

Description

```
EDD_ConnHandler(pthread_t &ThreadID, EDD_ApplInit *ApplObj);
```

The constructor obtains the file descriptor from **ApplObj**. The file descriptor identifies the connection to be handled.

```
void threadMain(EDD_ApplInit *ApplObj);
```

This is the main routine that waits for a request coming from the message queue. After it receives a request, it invokes a corresponding transaction member function. It also manages to send connection ACK/NACK, disconnection ACK and terminates the thread when a connection is relinquished.

```
virtual void applInitialize(EDD_ApplInit *ApplObj){}
```

You can redefine this function to perform the application initialization. This function is invoked when the thread is just up.

```
virtual int connectReq(void) = 0;
```

You must redefine this function. It is invoked from **threadMain** whenever the NEP requests a connection to a network element or a remote server. Typically, you call **getConnectParam** to retrieve the connection parameters defined in **tbl_comm_param** and then establish the connection. This function returns **ASC_Succeed**, if a connection is established successfully, otherwise, it returns **ASC_Fail**. When it returns **ASC_Fail**, the connection to NEP is removed and the thread is terminated.

```
char *getConnectParam(char *ParamLabel);
```

This function is provided to retrieve the connection parameters defined in **tbl_comm_param**. The **ParamLabel** is defined in **tbl_comm_param**. It returns a pointer to the value of the parameter or a **0** if the parameter is not defined.

```
virtual int processDataReq(void *Data, int Len) = 0;
```

You must redefine this function. It is invoked whenever the NEP requests to send data to or perform some operation on a network element or a remote server. The argument **Data** points to data coming from NEP and **Len** is the length of the data in byte. This function returns **ASC_Fail**, if it detects that the connection to the network element is unavailable. This event triggers the thread to close the connection to the NEP and exit.

```
void sendDataToNEP(void *Data, int Len);
```

You can use this function to send data back to the NEP. Typically, you forward the data coming from a network element or remote server to the NEP.

```
virtual void disconnectReq(void) = 0;
```

You must redefine this function. It is invoked whenever NEP requests to close the connection to a network element which is handled by this object. Once this function returns, the thread closes the connection to NEP and exits.

```
void setApplData(void *ApplData){ m_ApplData=ApplData; }
void *getApplData(void) { return m_ApplData; }
virtual void deleteApplData(void *ApplData) = 0;
```

You call these three functions to save the data specific to the application. Typically, this data is used between two calls to **processDataReq** or the data needed by the application during the thread life time. It is the application's responsibility to coordinate the use of this data field for different purposes.

You must redefine the **deleteApplData** function, since **setApplData** and the destructor calls this function to delete the application data. If you delete the data from outside of this function, you must set **m_ApplData** to **0**.

```
void cleanup(void);
```

This function cleans up all spaces allocated for the application to execute RPCs.

Three public member attributes are used by the framework and the application to log diagnostic messages, generate events and retrieve configuration variables.

All functions in the private section are used to handle the connections to NEP.

B

Oracle Execution Examples

The following examples demonstrate the execution of Oracle functions through the SQL*Plus client utility. These samples cover the general methods of function invocation. The Oracle login user requires execute permission on the function invoked; normally, the login user is the owner of the object (for example, the SARM database user).

Example 1

A function is invoked with no input or output arguments.

```
SQL> var retval number;
SQL> exec :retval := SSP_del_csdl_defn
```

Example 2

A function is invoked with several input arguments which are specified positionally. Note that empty strings are denoted as a <space> character if the parameter is required. If the parameter is optional, the null string can be denoted by two consecutive single quotes without a <space> character.

```
SQL> var retval number;
SQL> exec :retval := SSP_new_csdl_defn('M-CREATE_BUS_LINE', 'Y', 82, ' ', ' ', ' ');
```

Example 3

This example illustrates how to add or delete information from a database. The following example clears out and then adds several rows into tbl_csdl_config.

```
SQL> set serveroutput on
SQL> var retval number

SQL> prompt Removing CSDL Definitions from the SARM

SQL> exec :retval := SSP_del_csdl_defn;

SQL> prompt Adding CSDL Definitions to the SARM

SQL> exec :retval := SSP_new_csdl_defn ('C_NEW_FLAT_LINE', 'Y', 60, '', '', 'Add flat-
rate line');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_CIDB', 'Y', 65, '', '', 'Add Always-on
CID Block');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_DNY_IC', 'Y', 65, '', '', 'Deny
incoming calls');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_DNY_TOLL', 'Y', 65, '', '', 'Deny toll
calls');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_CWT', 'Y', 80, '', '', 'Add Call
```

```
Waiting');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_CID', 'Y', 80, '', '', 'Add
Caller ID');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_ACB', 'Y', 80, '', '', 'Add
Repeat Dial--*66');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_AR', 'Y', 80, '', '', 'Add Return
Call--*69');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_SCS', 'Y', 80, '', '', 'Add Speed
Call Short');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_3WC', 'Y', 80, '', '', 'Add 3-Way
calling');

SQL> exec :retval := SSP_new_csdl_defn ('C_ADD_CFW', 'Y', 80, '', '', 'Add Call
Forward');
```

Example 4

A function is invoked with input arguments that are bound by parameter name. Optional arguments are not passed. The order of the arguments is not relevant when binding by parameter name.

```
SQL> var retval number;
SQL> exec :retval := SSP_new_csdl_defn(csdl_cmd=>'M-CREATE_BUS_LINE',
csdl_level=>82,rollback_req=>'Y');
```

Example 5

All the previous examples may be run within a PL/SQL block, as shown here:

```
SQL> declare retval number;
2> begin
3> retval := SSP_new_csdl_defn('M-CREATE_BUS_LINE', 'Y', 82, ' ', ' ', ' ');
4> end;
5> /
```

Example 6

A function with a cursor result set is invoked. This case is the most complex since the cursor must be processed before SQL*Plus can view the result set. You require the definition of the return cursor, defined in the database package object (in this example, SarmPkg). The execution and results processing is performed within a PL/SQL block.

```
SQL> set serveroutput on
SQL> declare
  2     retcode integer;
  3     rc1     SarmPkg.SSP_list_csdl_defn_1;
  4     cur_rc1 SarmPkg.SSP_list_csdl_defn_rt1;
  5     csdl    varchar2(25) := '&csdl';
  6  begin
  7     retcode := SSP_list_csdl_defn ( rc1, csdl );
  8     dbms_output.put_line('Return code: ' || retcode);
  9     if rc1%isopen then
 10        loop
```

```
11          fetch rc1 into cur_rc1;
12          exit when rc1%notfound;
13          dbms_output.put_line('csdl_cmd = '||cur_rc1.csdl_cmd);
14          dbms_output.put_line('rollback_req = '||cur_rc1.rollback_req);
15          dbms_output.put_line('csdl_level = '||cur_rc1.csdl_level);
16          dbms_output.put_line('fail_event = '||cur_rc1.fail_event);
17          dbms_output.put_line('complete_event = '||cur_rc1.complete_event);
18          dbms_output.put_line('description = '||cur_rc1.description);
19       end loop;
20       close rc1;
21    end if;
22 end;
23 /
```

Example 7

In the following example, a Korn shell wrapper enables you to set values before running the script. The following example sets the diag level for all of the servers in one place, and change the name of the SRP client in all places at the same time.

```
# CTRL Tables:  tbl_appl_proc
#  tbl_component
#
#        File:  svr_cfg
#
#     Purpose:  To define the ASAP servers, and allow use of the
#  Class A start scripts.
#
# Stored Procedure Parameter Format:
# CSP_del_appl:
#    *Application Code (Server Name)
# CSP_new_appl:
#     Start Sequence
#     Server Type (M-Master Control, S-Server, C-Client)
#     Application Code (Server Name)
#     Control Server
#     Auto-Start (Y/N)
#     Program (name of executable in $PROGRAMS)
#     Diagnostic level
#     Diagnostic file name
#     Description of server
#    *Server type (ADM, CTRL, MASTER, SARM, SRP, OTHER)
# CSP_del_component:
#    *ASAP Territory
#    *ASAP System
#    *Server Name
# CSP_new_component:
#     ASAP Territory
#     ASAP System
#     Server Name
#
# * indicates an optional parameter
#
#

scr=$(whence $0)

# Get database password
CTRL_PASSWORD=$(GetPassword $CTRL_USER 2)
```

```
# Define Local Servers
LOC_SRPC=SRPC$ASAP_ENV

# Define default diagnostic level
DIAG_LEVEL=LOW


sqlplus -s $CTRL_USER/$CTRL_PASSWORD <<HERE | grep -v "successfully completed"
set serveroutput on
var retval number


prompt Removing the ASAP Applications

exec :retval := CSP_del_appl;


prompt Defining the ASAP Applications in Territory $ASAP_TERRITORY, Local
System $ASAP_SYSTEM

exec :retval := CSP_new_appl (1, 'M', '$CTRL', '$CTRL', 'N', 'ctrl_svr',
'$DIAG_LEVEL', '${CTRL}.diag', 'Master Control Server', 'MASTER');
exec :retval := CSP_new_appl (2, 'S', '$SARM', '$CTRL', 'Y', 'sarm',
'$DIAG_LEVEL', '${SARM}.diag', 'SARM Server','SARM');
exec :retval := CSP_new_appl (3, 'S', '$SRP', '$CTRL',  'Y', 'srp_emul',
'$DIAG_LEVEL', '${SRP}.diag', 'SRP Emulator','SRP');
exec :retval := CSP_new_appl (4, 'S', '$ADM', '$CTRL',  'N', 'admn_svr',
'$DIAG_LEVEL', '${ADM}.diag', 'Administration Server','ADM');
exec :retval := CSP_new_appl (5, 'S', '$NEP', '$CTRL',  'Y', 'asc_nep',
'$DIAG_LEVEL', '${NEP}.diag', 'NEP Server','NEP');
exec :retval := CSP_new_appl (10, 'C', '$DAM', '$CTRL',  'N', 'run_asapd',
'$DIAG_LEVEL', '${DAM}.diag', 'ASAP Daemon','daem');
exec :retval := CSP_new_appl (13, 'C', '$JSRP', '$CTRL', 'N', 'srp.ear',
'$DIAG_LEVEL', '${JSRP}.diag', 'Java SRP Server','SRP');
exec :retval := CSP_new_appl (14, 'C', '$LOC_SRPC', '$CTRL', 'Y',
'runSrpClient', '$DIAG_LEVEL', '${LOC_SRPC}.diag', 'Java SRP Client','SRP');


prompt Removing the ASAP Components

exec :retval := CSP_del_component;


prompt Defining the ASAP Components for Territory $ASAP_TERRITORY, Local
System $ASAP_SYSTEM

exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$CTRL');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$SARM');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$SRP');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$ADM');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$NEP');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$DAM');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM', '$JSRP');
exec :retval := CSP_new_component ('$ASAP_TERRITORY', '$ASAP_SYSTEM',
'$LOC_SRPC');


HERE
```

C

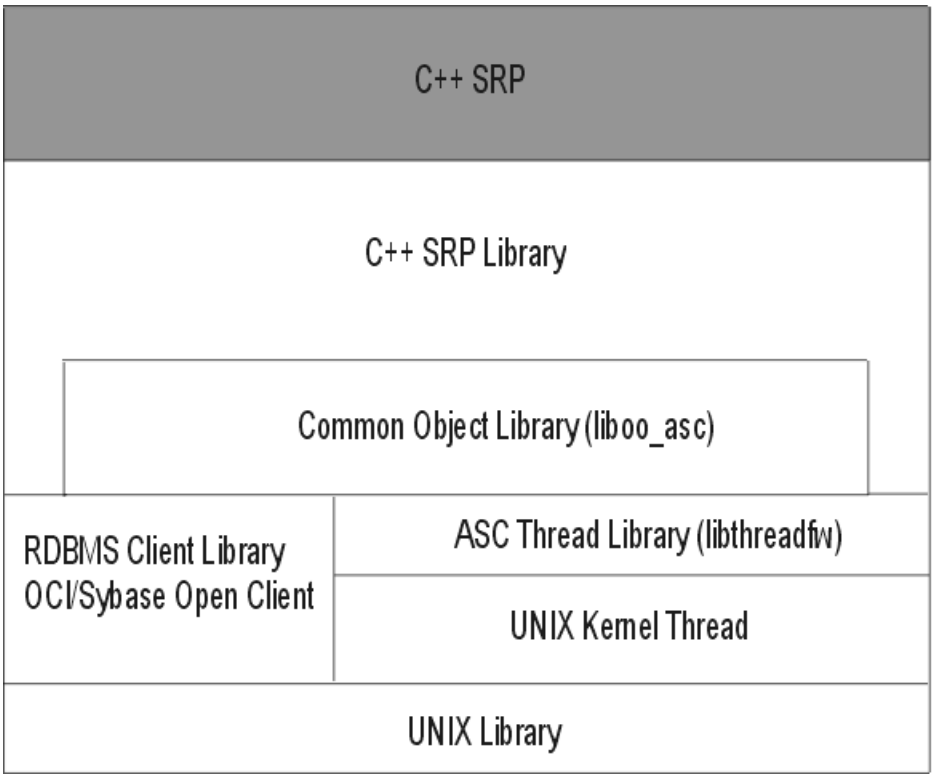
C++ SRP API Template Design

This chapter is intended for C++ SRP API developers and designers who have no prior experience developing the C++ SRP API. The following sections are included:

API library structures

Figure C-1 illustrates the APIs that can be linked to a C++ SRP API.

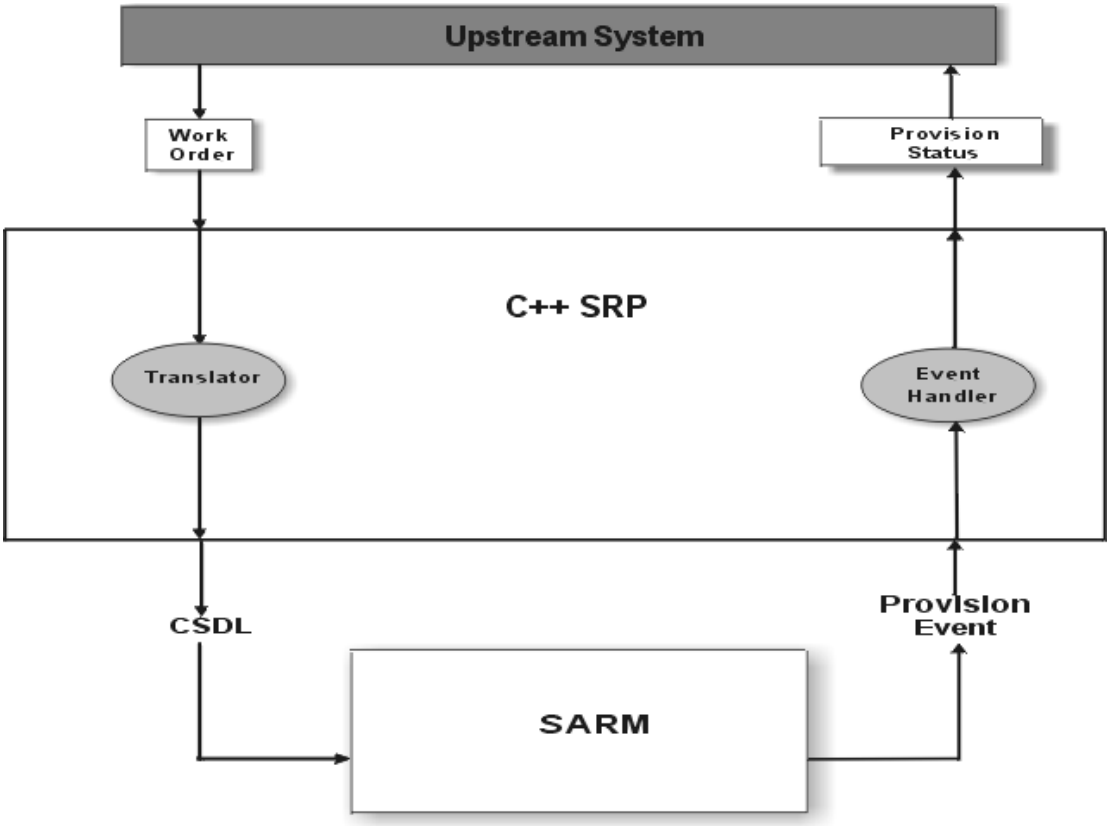
Figure C-1 C++ SRP APIs



The C++ SRP API provides the following capabilities:

- Interface protocols with external systems.
- Functionalities of the C++ SRP API library
- Controls data transfer between upstream systems (submitting work order to ASAP for provisioning) and the SARM.

Figure C-2 Data Transfer Through the C++ SRP API



Fundamental tasks performed by all C++ SRP API:

1. Converts work orders containing service requests in their native format to CSDL and sends work order to the SARM for provisioning.
2. Returns the work order provisioning status passed from the SARM to the upstream system.

C++ SRP API library

The C++ SRP API library is a library in ASAP to let the user applications interface with SARM. In addition, the library will help the users to develop SRP in ASAP with an object-oriented interface. It provides the POSIX thread interfaces.

The C++ SRP API library provides the following functionality:

- OO interface to generate and submit work orders.
- OO interface to manipulate work orders.
- OO interface to process work orders.
- OO interface to retrieve work order information in ASAP.
- ASAP high availability.
- Reliable communication between SRP and SARM.

Common object library (liboo_asc)

The liboo_asc library provides you with the following functionality:

- Diagnostic message logging facility to the application programmers.
- System event generation.
- Application configuration parameter determination.
- Remote Procedure Calls API.
- Network connection management.

ASC thread library (libthreadfw)

Thread Framework provides an object-based interface for the application use threads. Other applications, which use UNIX threads, typically, POSIX thread and DCE threads can use this interface.

C++ SRP API components

Non-interfering tasks are separated into threads in the C++ SRP. Different threads (non-interfering task) can process their own task at the same time to improve performance (increase parallel processing). The threads in C++ server are:

Work order submission

- **Receiver** – Establishes connection and receives service request data from upstream system.
- **Translator** – Translates service request in native format to CSDL command, Work Order and Parameters.
- **SARM Driver** – Submits ASAP work order to SARM for ASAP provisioning.

Event notification

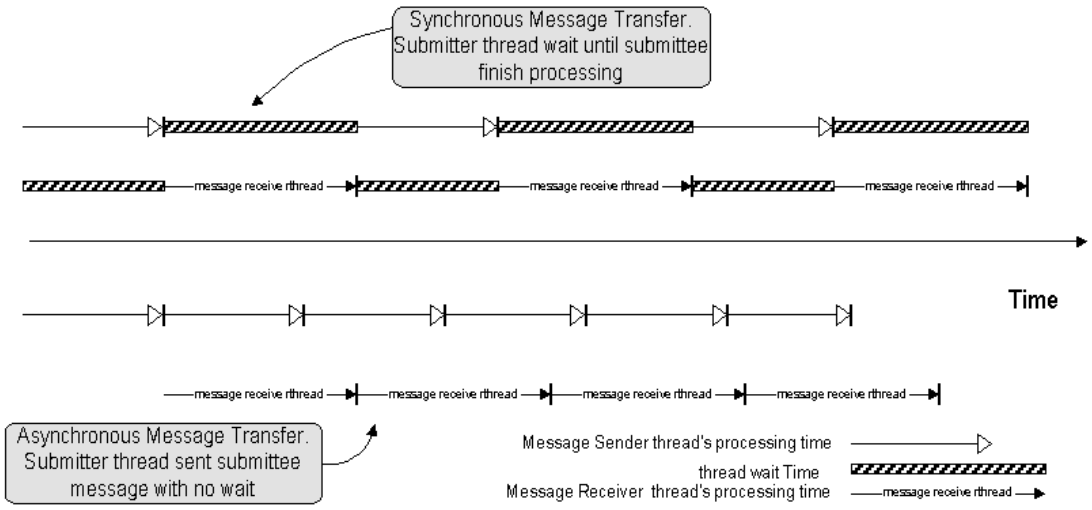
- **Work Order Manager** – Accepts provisioning event from SARM. Sends event handling message to event handler.
- **Event Handler** – Passes appropriate data to sender thread for different provisioning events.
- **Sender** – Connects and returns the provisioning status to the upstream system.

Communication between threads

Threads communicate with each other using the message queue.

- **Synchronous Message Transfer** – Message sender waits for the message receiver to finish processing (Sequential).
- **Asynchronous Message Transfer** – Message sender does not wait for message receiver to finish processing (Parallel).

Figure C-3 Communication Between Threads



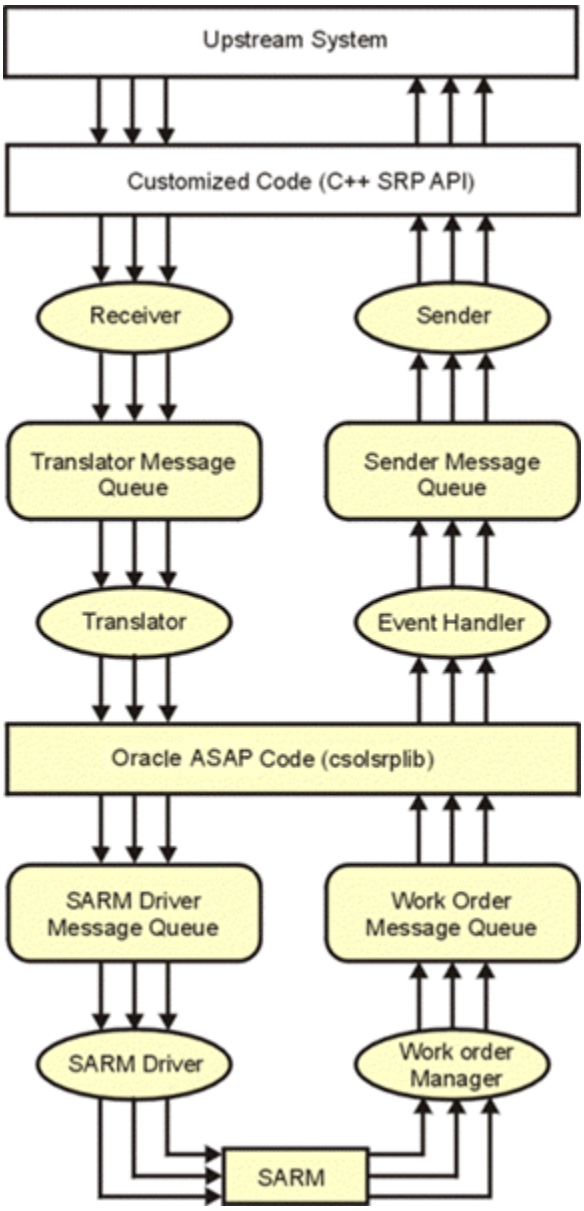
The following message queues are used in the communication between threads:

- **Translator message queue** – The receiver thread sends a pointer to a work order structure containing a service request in native format to a translator thread.
- **SARM Driver message queue** – The translator thread sends a pointer to the ASAP work order structure containing the CSDL commands to the SARM Driver thread.
- **Work Order Manager message queue** – The work order manager thread sends the provision status to the event handler thread.
- **Sender message queue** – Event handler passing message structure returns to Upstream system to sender thread.

Multiple instances of threads

Multiple instances of the same type of thread can process multiple work orders in parallel. The number of thread instances can be configured in the configuration file to obtain optimal results. To improve performance, multiple work orders can be processed at the same time. The complexity of the C++ SRP increases due to the synchronization between threads and work order dependency.

Figure C-4 Multiple Threads

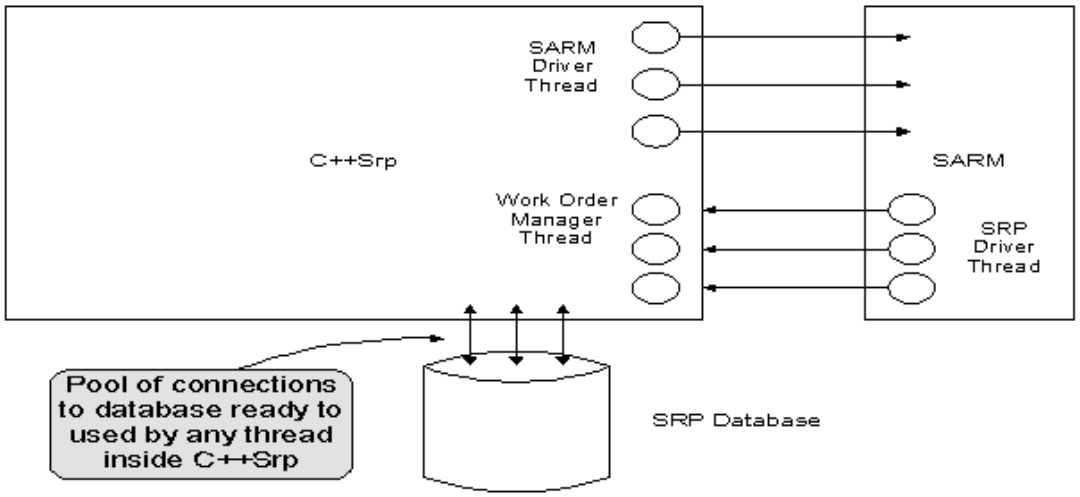


Communications with ASAP internal systems

The C++ SRP maintains constant connection with the following internal systems:

- **SARM** – C++ SRP API submits CSDL commands and obtains provisioning status.
- **SRP database** – C++ SRP API obtains static information from the SRP database and may use the database to store work order information.

Figure C-5 C++ SRP API Communication



Communication between C++ SRP API and SARM

- SARM Driver thread and the Work Order Manager thread in the CsolSrp Library, and the SRP Driver thread in SARM, handle the connections between the C++ SRP API and SARM.
- Sybase Open Client RPC handles the data transfer between the C++ SRP API and SARM. The Socket RPC message goes from SARM to the C++ SRP API.

Communication between C++ SRP API and SRP database

- C++ SRP API maintains a pool of connections with the SRP database using the Sybase Open Client connection routines.
- Any thread in the C++ SRP API can allocate connection from the pool using **ClntProcMgr** Class.
- RPCs are preferred over SQL statements to transfer data between the C++ SRP API and the SRP database.

Upstream system interface

This section describes upstream system interface.

Protocol

The Upstream System interface protocol consists of:

- Communication between the C++ SRP API and Upstream systems
- TCP/IP Sockets
- Connection Verification

Communication between C++ SRP API and upstream systems

- Sender thread and Receiver thread in each C++ SRP API handles the connections between C++ SRP API and upstream system.
- Connection method is site specific.
- Commonly used communication protocol, for example, TCP/IP (socket), Open Client/Server Language Communication, SNA LU 6.2).
- Routines must be thread-saved with the Sybase multithread library.
- Synchronous communication is used, for example, acknowledgement required for data transfer.

TCP/IP sockets

- Implementation of TCP/IP requires a new thread (Connection Handler thread) in the C++ SRP API server.
- Connection Handler Thread [server] constantly waits for connection from the upstream system [client].
- IP address and port number belong to the upstream system, and the C++ SRP API is defined in command line or input file.
- To create a socket, the Connection Handler Thread uses **listen**.
- To wait for I/O from the upstream system and connection socket, **Accept** is used.
- To handle the data transfer between C++ SRP API and upstream system, the Connection Handler thread spawns the Receiver thread.
- **Read** and **write** are used for data transfer between receiver and upstream system.

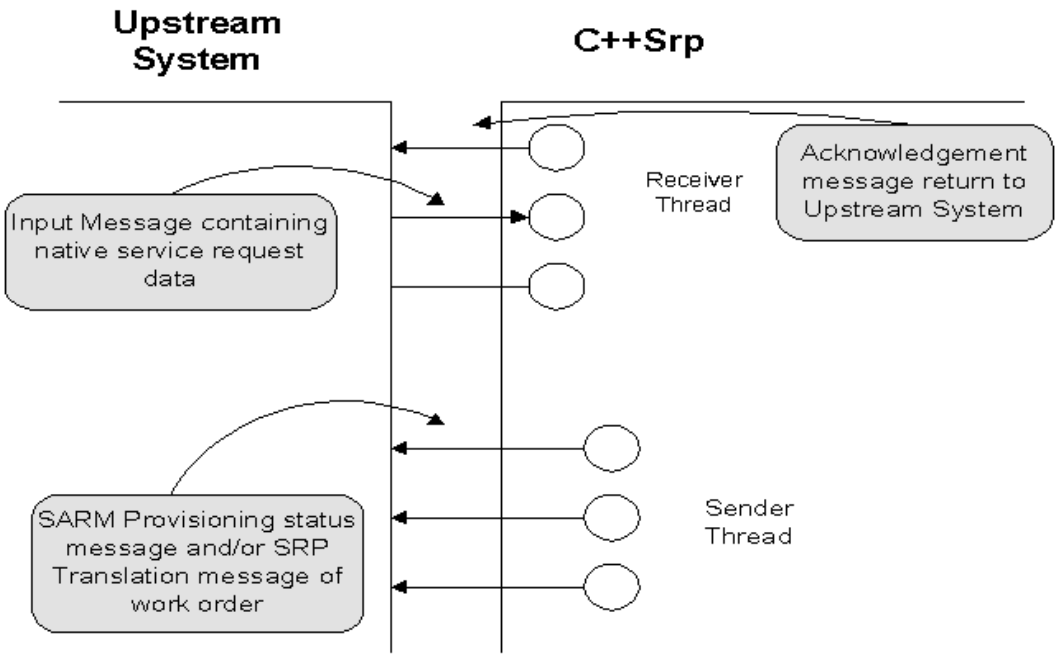
Connection verification

The Receiver thread verifies the upstream system connection before accepting data from connection (for example, disconnect connection if limit reached).

Data format

The data format used must be the same for both the upstream system and customized SRP.

Figure C-6 C++ SRP API Data Format



Input message from upstream system

The message from the upstream system received by receiver thread, for example, service request.

Label value pair:

- Linear representation of the work order.
- Simple format, hard to represent complex work order.
- API function **get_name_value( )** can extract the work order with the format **label=value;\n**.

Return message to upstream system

- Message that is returned to the upstream system concerns provisioning status of work orders.
- Message can also be SARM submission status of Translator in the asynchronous processing.
- Work Order identifier is included as part of the message.

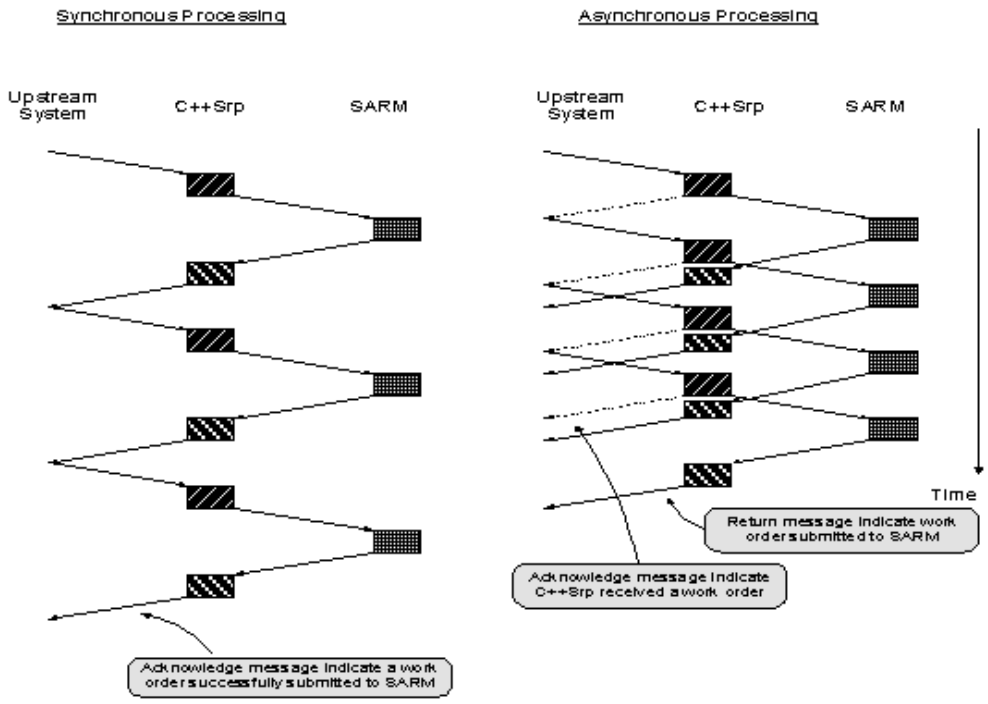
Synchronous processing

1. The C++ SRP API returns the acknowledgment to the Upstream system after the work order submits to SARM.
2. Upstream system submits the next work order after it receives an acknowledgment from the C++ SRP API.

Asynchronous processing

1. The C++ SRP API returns acknowledgment to the Upstream system after the C++ SRP API successfully receives a message from the Upstream system.
2. Upstream system submits the next work order after it receives an acknowledgement from the C++ SRP API.
3. The C++ SRP API translates the work order and submits the work order to SARM. During this time, the current work order has no persistence (for example, if the C++ SRP API system goes down, the work order cannot be recovered).
4. The C++ SRP API returns to the Upstream system after the work order successfully submits to SARM.
5. Upstream system needs to maintain the work order submitted to the C++ SRP API until the C++ SRP API successfully submits the work order to SARM.

Figure C-7 Synchronous and Asynchronous C++ SRP API Processing



If Upstream system can persist, the work order is submitted to the C++ SRP API before the work order is submitted to the SARM, therefore work order dependency is not important or is maintainable. The asynchronous processing is recommended because of its faster throughput.

Table C-1 Processing characteristics

Synchronous	Asynchronous
Receiver and Translator thread process in sequence.	Receiver and Translator thread process in parallel.
SRP return SARM submission status to upstream through receiver thread.	C++ SRP API return SARM submission status through sender thread (other connect to upstream system).
Slow throughput.	Fast throughput for work order submission.
Simpler Implementation	Harder implementation
Easy re-submission of work order during system failure.	Hard implementation of re-submission of work order during system failure.
Upstream system does not need to maintain work order once acknowledgement received from SRP.	Upstream system needs to maintain work order after acknowledgement received, until the SRP returns stating work order successfully saved in SARM.
Work Order dependency preserves because work order submits to SARM in the same sequence work order submitted from Upstream.	Work Order dependency needs to manage by C++ SRP API because work order might not submit to SARM in the same sequence work order submits from upstream system.

If throughput is not as important, synchronous processing should be used for simple implementation.

Single and multiple connections

If the volume of depending work order is very low and the upstream system can handle multiple connections, you should use multiple connections for increased parallel processing (better speed).

If the volume of depending work order is high, you should use a single connection for easier implementation.

Table C-2 Single and multiple connections

Single Connection	Multiple Connections
One receiver thread in C++ SRP system submits work order to translator thread.	Multiple receiver threads submit work order message to translator thread.
Work order dependency can be maintained if the submission sequence is the same as the dependency sequence.	Method handling work order dependency is required if the depending work order can be sent through a different connection.
Easier implementation.	Complex implementation.
Sequential processing.	Parallel processing.
Low throughput.	High throughput.
No backup connection if connection fails.	Backup connection available when one connection fails.

API libraries

The following API libraries are used to develop SRPs:

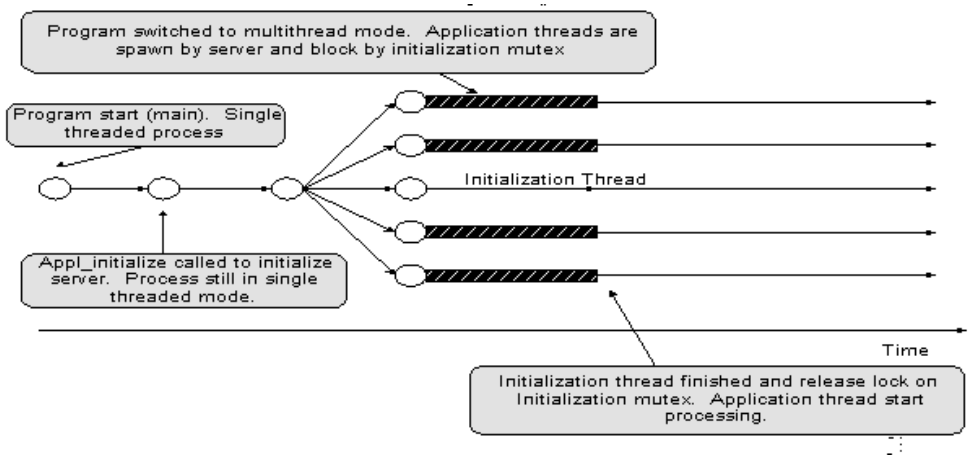
- **Common API** (liboo_asc) – Network connection handling functions.
- **Thread API** (libthreadfw) – Multithread Server routines.
- **C++ SRP API** (libCsolSrp) – CsolSrp required routines
- **Interpreter API** (libinterpret) – Interpreter routines

Main( )

The program starts up in single-thread mode. The initialization code in **main( )** is executed while the process is still in single-thread mode. After initialization, the process turns multi-thread.

Application threads are spawned and wait for the initialization thread to release the initialization mutex. When the initialization thread has finished, it releases the initialization mutex. Application threads can start its process.

Figure C-8 Multithread Server Initialization



SRP_initialize

Calls **SRP_initialize()** in **appl_initialize()** before other initialization routines. Add functionality that is generic to all SRP servers.

SRP_initialize() creates the following threads:

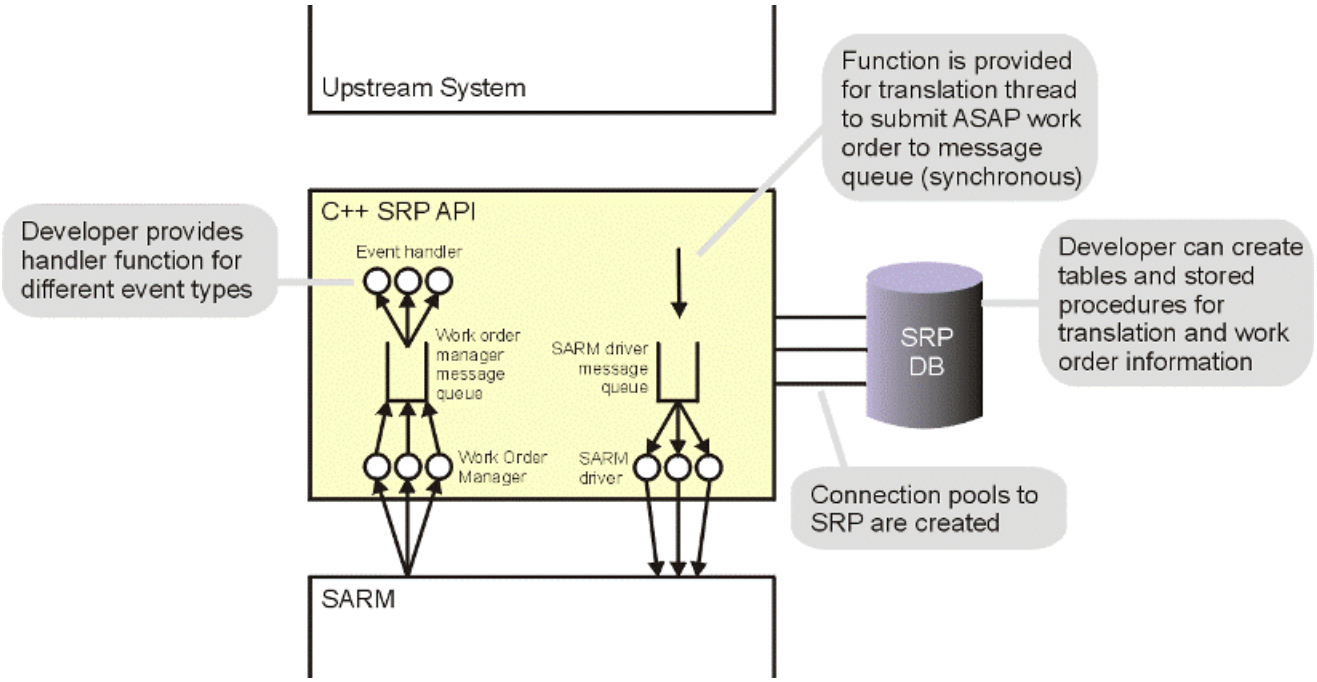
- SARM Driver thread
- Work Order Manager thread
- part of event handler thread

SRP_initialize() creates the following message queues:

- Work Order Manager
- SARM Driver

Figure C-9 shows the components that are added to C++ SRP API server after call to **SRP_initialize**.

Figure C-9 SRP_initialize Processing



You must develop code that:

- Interfaces with the Upstream system (receiver thread and sender thread).
- Translates a service request in native format to CSDL (Translator thread).
- Submits an ASAP work order containing CSDL to SARM Driver message queue.
- Handles a provisioning event from SARM by defining the event handling function.

C++ SRP threads

The following section describes the threads in the C++ based SRP server:

- Receiver
- Translator
- Event Handler
- Sender

Receiver

The following tasks are performed by the receiver:

- Connects to the Upstream system using the protocol agreed with upstream system.
- Validates the connection for security purposes.
- Waits for the work order containing the service request in the native format from upstream system.
- Checks the dependency of work order.
- Allocates the translator message structure, stores the work order in the native format and provides the field containing the translator return status (synchronous).
- Submits the pointer to the translator message to the Translator message queue.
- Waits for the return status from the translator.
- Formats the acknowledgement message.
- Returns the acknowledgement message to the upstream system.
- Waits for the next request from the upstream system.

Connecting with the upstream system

This is site specific. The common protocols used are TCP/IP sockets.

Multiple concurrent connections can be accepted from the upstream system, that is, multiple instances of the receiver thread. However, the dependency between work orders submitted from different connections must be managed properly by the receiver before it is sent to the translator message queue for translation.

To benefit from multiple connections, multiple translator threads and the SARM driver must be present to obtain maximum performance.

Verifying incoming message

Verification can be performed after a formatted data string is successfully received from the upstream system. The purpose of the verification is to check if the message is in the appropriate format that is understandable by the SRP or any other restriction without delimiting the raw message.

Synchronous processing

- The receiver thread submits the message to translator message queue with wait.
- Translator thread wakes up the receiver thread after the work order successfully submits to SARM.
- Translator message contains the field to store the SARM submission status. The receiver has the responsibility to deallocate the translator message structure after translator returns.
- Receiver returns the acknowledgment to the upstream system and waits for the work order from the upstream system.

Asynchronous processing

- Receiver thread submits the message to the translator message queue with no wait.

- Translator thread process in parallel with the receiver thread.
- Receiver thread returns the acknowledgement to the upstream system after it submits the message to the translator thread.
- After the work order submits to SARM, the translator is responsible for freeing memory allocated for the translator message.

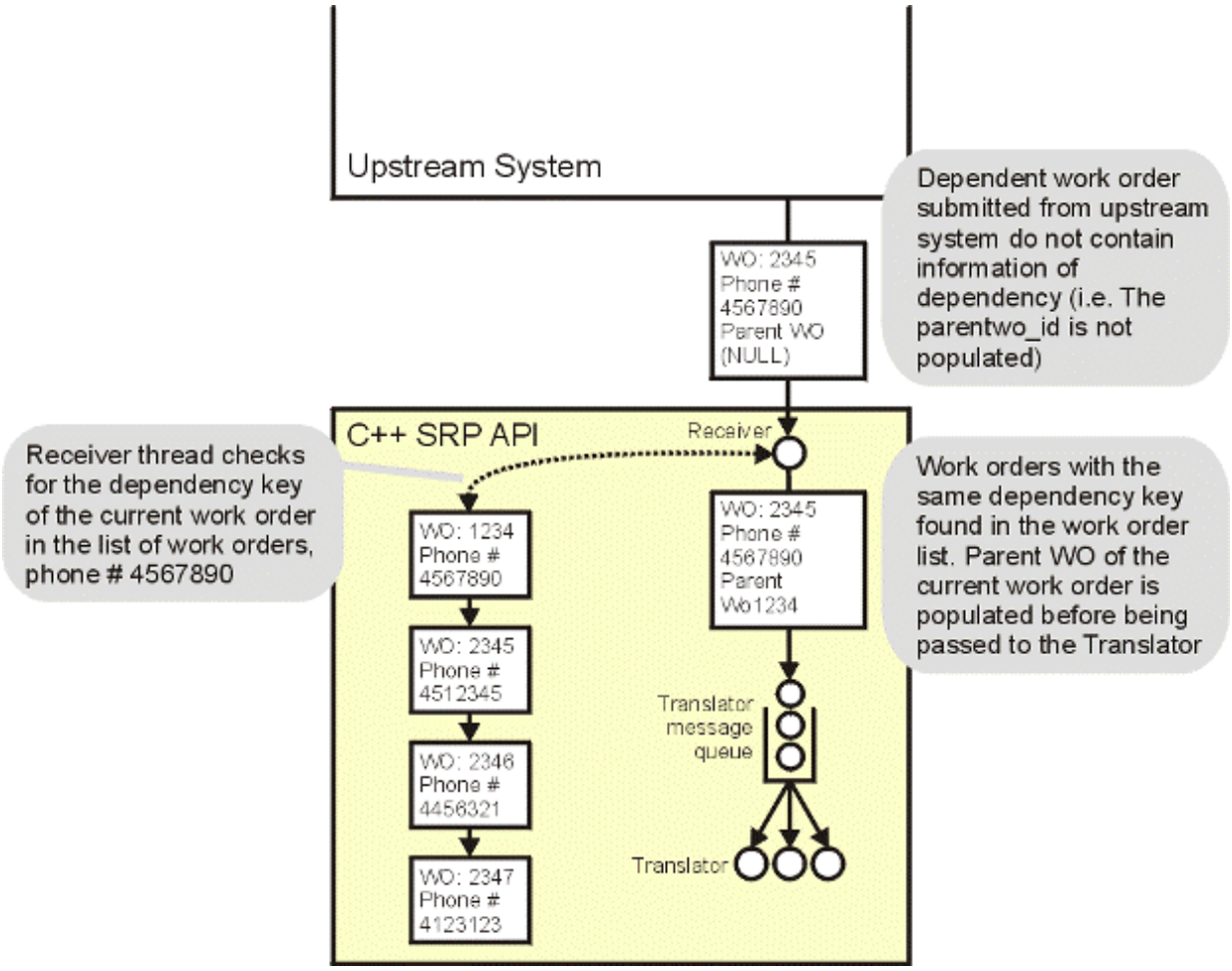
Thread examples

This section provides thread examples.

Single connection, asynchronous processing (work order dependency)

- One Receiver thread and multiple Translator thread is used.
- Dependent work orders are submitted to the C++ SRP API in sequence.
- Work orders are submitted to the SARM in parallel when asynchronous processing. Child work orders might be submitted to SARM and processed by SARM before parent work orders.
- Receiver thread maintains the work order list order by a key value that determines dependency.
- Work order received from the upstream system will check the list if the key is found. The parent work order of the current work order is updated to the work order already in the list. Then the work order is submitted to translator.
- When SARM returns successful provision status of the work order, the work order will be deleted from the list.

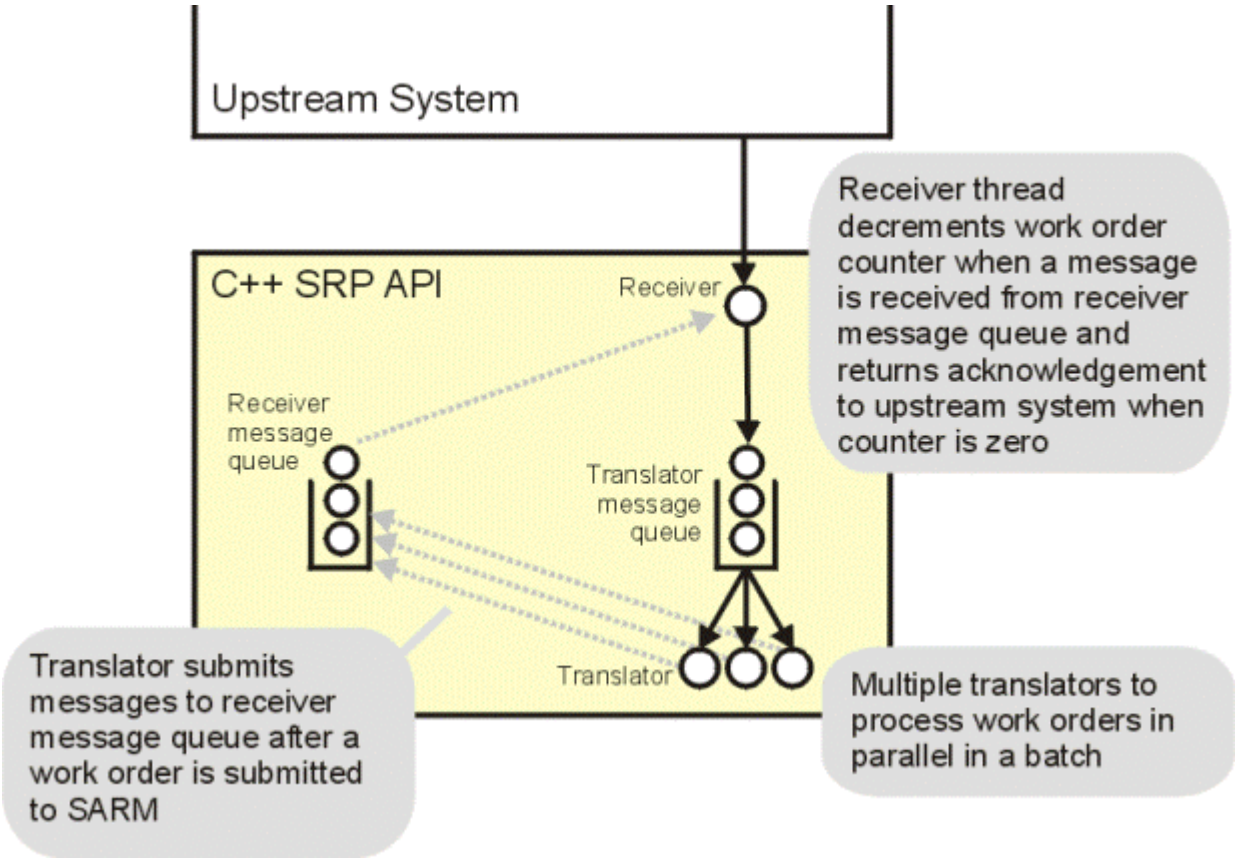
Figure C-10 Single Connection, Asynchronous Processing (Work Order Dependency)



Single connection, synchronous processing (batch submission of work order)

- One Receiver thread and multiple Translator thread are used.
- Multiple work orders are submitted to the C++ SRP API in batches within one data transfer from the upstream system.
- Receiver maintains a counter of the number of work orders and a message queue (Receiver message queue).
- After the receiver thread submits all work orders of a batch to the Translator thread, it waits for the message from the Receiver message queue.
- Translator thread sends a message to the Receiver message after finish submitting one work order to SARM.
- Receiver thread decrements the work order counter when it receives one message from the translator. Once the counter reaches zero, the Receiver returns an acknowledgement message to the upstream system and waits for the next batch of work orders.

Figure C-11 Single Connection, Synchronous Processing (Batch Submission of Work Order)



Translator

The following tasks are performed by the translator:

- Receives the message from the translator message queue
- Constructs the work order (intermediate structure) from the service request passed in from the upstream system.
- Translates the work order into the ASAP work order structure and sends it to SARM.
- Synchronous translation – Updates the return status inside the translator message, and wakes up receiver thread.
- Asynchronous translation – Allocates to the sender message structure and sends the structure to the sender message queue asynchronously.

The following steps are required to return the translation status to the upstream system:

Table C-3 Synchronous vs asynchronous processing

Synchronous Processing	Asynchronous Processing
Update the return status in the Translator Message structure.	Allocate and populate the Sender Message structure with the status of translation.

Table C-3 (Cont.) Synchronous vs asynchronous processing

Synchronous Processing	Asynchronous Processing
Wake up the Receiver Thread.	Send a message to Sender Message queue without waiting (asynchronously).
Receiver thread will lookup the translation status in the Translator Message structure.	Translator frees the memory allocated for the Translator Message structure.
Receiver frees the memory allocated for the Translator Message structure.	Translator status will send through the connection established with Sender Thread.
Receiver will return the Translation status as ACK/NACK to the upstream system through the connection established with receiver	-

Event handling

This section describes event handling.

SARM events

At different stages of provisioning, SARM will notify the C++ SRP Work Order Manager thread. The Work Order Manager (handled by API), in turn, spawns threads to handle the event.

The **SRP_EventManager** manages a connection with SARM to receive work order events. When a **SRP_EventManager** thread starts up, it waits to receive a message, which is an event, from SARM. The message is constructed as a DU packet format, which is provided. The **SRP_EventManager** thread decodes the message and creates the **SRP_Event** object to the corresponding event type in the messsage. The thread calls the corresponding event handler for the event with the **SRP_Event** object. After the event handler is done, it returns to receive an event. If an event is the **kick_start**, the thread does not create a **SRP_Event** object. It just sends **CS_TRUE** back to the SARM.

The following are the possible events and the sequence of events for a work order:

1. SRP_WO_ESTIMATE_EVENT
2. WO_STARTUP_EVENT
3. WO_SOFT_ERROR_EVENT
4. WO_NE_UNKNOWN_EVENT
5. WO_ROLLBACK_EVENT
6. WO_TIMEOUT_EVENT
7. WO_FAILURE_EVENT
8. WO_COMPLETE_EVENT

The sequence of events is guaranteed for a work order. For example, a WO_FAILURE event will not occur before a WO_STARTUP event.

Actions performed by event handler

All action methods send a diagnostic message for performance testing to indicate in what is the status of the work order and then calls the appropriate RPC to update the SRP database to reflect the new conditions of the work order.

- EventHandler::softErrorHandler().
- EventHandler::woEstimateHandler()
- EventHandler::woStartHandler()
- EventHandler::woRollbackHandler()
- EventHandler::neUnknownHandler()
- EventHandler::woBlockHandler()
- EventHandler::woTimeOutHandler()
- EventHandler::neAvailHandler()
- EventHandler::neUnavailHandler()
- EventHandler::woAcceptHandler()
- EventHandler::woCompleteHandler()
- EventHandler::woFailureHandler()

Sender

Sender manages the connection with the upstream system to return provisioning information to the upstream system. The Sender thread can receive a message from an event handler or translator (asynchronous processing).

The following tasks are performed by the Sender thread:

- Establishes the dedicated connection with the upstream system.
- Waits for a message from sender message queue.
- Formats the sender message.
- Sends the message to the upstream system.
- If the sender message comes from the event handler, it wakes up the event handler thread.
- If the message comes from the translator, it deallocates the sender message.

Sender message

Wo_id must be part of the sender message to allow the upstream system to recognize which work order the message belongs to.

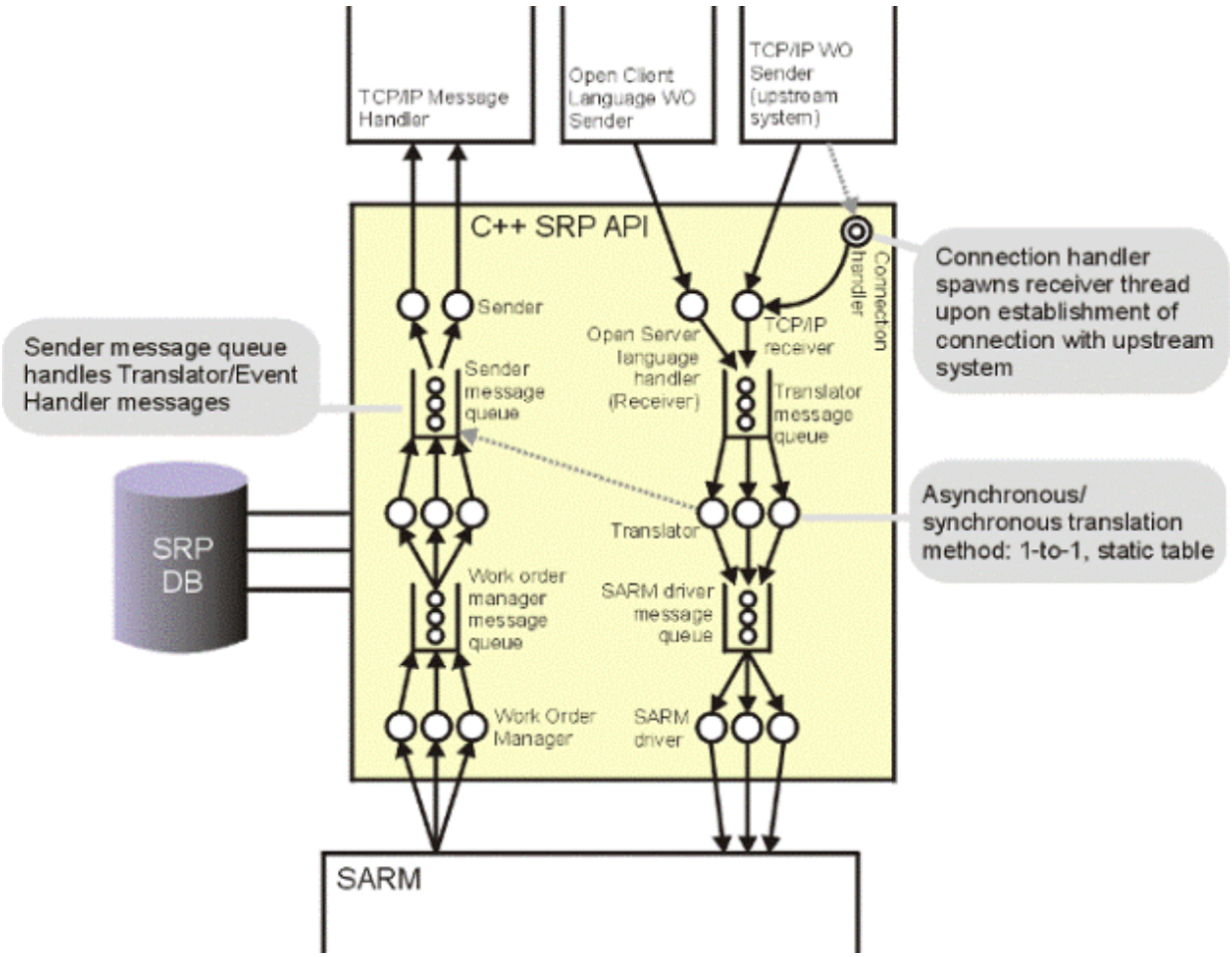
If the Translator sends a message to the Sender, all Sender Messages should include the type of message (for example, translator or event message) because the translation message is different from the event message.

C++ SRP API specification template example

The C++ SRP API Template example contains the following:

- Support TCP/IP socket connection.
- Choose either Asynchronous or Synchronous Translation with compile option –DASYNC or –DSYNC respectively.
- If the static table or interpreter translation is used, the database table and function in the Translator Thread must be created and populated.

Figure C-12 Specification Template Example



Communication interface

This section describes the communication interface.

TCP/IP socket interface

The TCP/IP socket interface contains the following:

- Connection Handler
- Receiver

Connection handler

- One instance of this thread is spawned when the C++ SRP API server starts.
- Connection Handler thread will listen for the connection from the port specified in the ASAP configuration file.
- When the connection is detected, it can check the client connection information.
- Spawns the Receiver thread and passes in the socket descriptor of the client connection to handle the incoming data.
- Waits for the next connection.

Receiver

- Spawned by the connection handler once the connection is detected.
- Loops and does the following:
  - Reads the message header and message from the socket connected to the upstream system and populates the translator message structure.
  - If the connection drop is detected, it terminates the current receiver thread only.
  - If the connection drop is not detected, it verifies the message retrieved from the socket.

Table C-4 Compilation options

Compile with SYNC defined	Compile with ASYNC defined
Submits message to the translator synchronously.	Submits message to the translator asynchronously.
Waits for the translator to finish translation.	Returns ACK/NACK to the upstream system through the socket connected to the upstream system.
Retrieves the return status of translation from translator message.	-
Returns ACK/NACK to the upstream system through the socket connected to the upstream system.	-

Translator thread

- **Main** program spawns the translator threads and creates the translator message queue when the server starts up.
- Repeat the following:
  - Translator thread waits for the message from the receiver through translator message queue.
  - Translator parses out the service request in the native format and stores the tokens in the SRP work order structure.

- Creates **SRP_WO** object base on SRP work order structure.
- Translators submit **SRP_WO** to SARM using **submitWO**.

Table C-5 Compilation options

Compile with SYNC defined	Compile with ASYNC defined
Translator populates the translation message status passed in from the receiver thread.	Translator constructs the sender message with translator type.
Wakes up the receiver thread.	Submits the sender message to the sender message queue with no waiting.
-	Free memory allocated for the translator message.

Translation process

The C++ SRP API template supports translation methods. You must provide the mapping of the native service request to CSDL command.

1 to 1 mapping:

Direct mapping of the service request to the CSDL with no translation.

Static Table translation:

- Load data in the database tables into SRP memory using the RPC (function in SRP database).
- For each service request in SRP work order, look up the CSDL from the table in memory using **bsearch**.
- For every parameter of the CSDL, search for the appropriate parameters in the SRP WO and add to the ASAP WO.

Event handling

- Events – woStartupHandler, woCompleteHandler, woFailureHandler, etc.
- Populates the sender message.
- Retrieves the CSDL log from the SARM database.
- Submits the message to the sender with wait (synchronous transfer).
- Frees the sender message.

Sender thread

- C++ SRP API server spawns sender threads when the server starts up.
- Sender message can be an event message or a translation status message (if ASYNC is defined).
- Sender will format the return message based on the message type and sends it back to the upstream system.

Event message handling

The Event sender message is submitted synchronously. Upon completion, the Sender will wakeup the event handler thread.

Translation message handling

The Translator sender message is submitted asynchronously. The Sender can free memory after the message has been sent to the upstream system.

Upstream system

The following is the upstream system process.

WO submission

- Program will connect to C++ SRP API server.
- Read message files from a directory.
- Send data in the file to C++ SRP API.
- Print an acknowledgment message to standard output.
- TCP/IP.

Handling WO provision results

- TCP/IP version only – The program acts as server.
- Listen for C++ SRP API connections.
- Read the message from the connection socket with the C++ SRP API.
- Print message to standard output.

Configuration for C++ SRP API

To add a C++ SRP API to the system, the following steps are required:

1. Add the C++ SRP API to the SARM database (**tbl_asap_srp**).
2. Set auto-start to **N** in the ASAP start-up procedure (**tbl_appl_proc**).

D

API and Other Configuration Changes

This appendix outlines various API issues related to upgrading ASAP from ASAP 4.5 to later versions.

OSS through Java service activation API

The OSS through Java service activation API used in ASAP 4.5 was **JSR 89 v0.8**. The new Java service activation API is: **JSR 89 v1.0**

JVT API changes

Following list outlines the interface changes between the JSR 89 v0.8 and v1.0 API.

Table D-1 JVT API changes

-	ASAP 4.5	ASAP 4.6.x and later
REQUESTED_COMPLETION_DATE	-	javax.oss.order.OrderValue defines an attribute called REQUESTED_COMPLETION_DATE. If it is populated when the order is created, ASAP uses this attribute as the work order due date. If a client invokes JVTActivationSession.startOrderByKey(OrderKey key), the order is started immediately. In this situation it resets the REQUESTED_COMPLETION_DATE to be the current date time regardless of whether it was previously set or not.
JVTActivationSessionOptionOps	javax.oss.order.JVTActivationOption	javax.oss.order.JVTActivationSessionOptionOps.
JNDI Naming	System.<EnvironmentID>.ApplicationType.Activation.Application.1-0-4_5-ASAP.Comp.<ComponentName>	System.<EnvironmentID>.ApplicationType.ServiceActivation.Application.1-0;4-6;ASAP.Comp.<Component Name> System/<EnvironmentID>/ApplicationType/ServiceActivation/Application/1-0;4-6;ASAP/Comp/MessageQueue JVTEventTopic, to which the Java SRP sends events if clients make requests through JVT interface System/<EnvironmentID>/ApplicationType/ServiceActivation/Application/1-0;4-6;ASAP/Comp/JVTEventTopic XVTEventTopic, to which the Java SRP sends events if clients make requests through XML/JMS interface System/<EnvironmentID>/ApplicationType/ServiceActivation/Application/1-0;4-6;ASAP/Comp/XVTEventTopic
com.mslv.oss.activation.JVTXAcvtivationSession	queryAudit(QueryValue, String[]), queryServiceHistory(QueryValue, String[]), queryAsdlHistory(QueryValue, String[]) removed	Same functionality can be performed by queryManagedEntities(QueryValue, String[]) in JVTActivationSession interface

Table D-1 (Cont.) JVT API changes

-	ASAP 4.5	ASAP 4.6.x and later
JMS Message Headers	OSS_APPLICATION_DN -identifying the OSS application that has published the message. OSS_ORDER_TYPE - describing the type of order that has changed. OSS_EVENT_TYPE - describing what kind of event has been sent. OSS_API_CLIENT_ID -identifying the client that owns the order entity.	OSS_APPLICATION_DN, identifying the OSS application that has published the message. OSS_EVENT_TYPE, describing what kind of event has been sent. OSS_ORDER_TYPE, describing the type of order that has changed. OSS_API_CLIENT_ID, identifying the client that owns the order entity. Java SRP defines additional properties specific to ASAP: OSS_ORDER_PRIMARY_KEY, identifies an order's primary key, same as orderKey.getPrimaryKey(). This property applies to all messages sent by Java SRP, except the messages that contain multiple orders, in which case, the property will be set empty. OSS_ORDER_EXCEPTION, indicates the work order was completed with exceptions. Used by OrderCompleteEvent. Such exceptions are generally the result of a “Fail but Continue" status being returned to the SARM for one of the ASDLs on the work order. This field is set by the SARM and communicated to the relevant SRP, which then requests the exception details. The possible values include: • _ (Y) ASAP_WO_EXCEPTIONS-the work order completed with exceptions. • _ (N) ASAP_WO_NO_EXCEPTIONS-the work order completed without any exceptions. • OSS_ORDER_ESTIMATE, the estimated amount of time (in seconds) for the work order to be completed by the SARM. This is the total time from the order initially being received by the SARM, to it being provisioned by the SARM. Used by OrderEstimateEvent • OSS_ORDER_MISC, the miscellaneous information. Used by OrderEstimateEvent • OSS_ORDER_SERVICE_NE, identifies the Network Element that is not known to the SARM. Used by OrderNEUnknownEvent • OSS_ORDER_CURRENT_STATE, identifies the current state of the order. Used by OrderStateChangeEventOSS_ORDER_REASON, describing the reason of the order state change. Used by OrderStateChangeEvent
IllegalStateException	javax.oss.order.IllegalStateException	javax.oss.IllegalStateException.

Table D-1 (Cont.) JVT API changes

-	ASAP 4.5	ASAP 4.6.x and later
javax.oss.order.JVTActivationSession	createOrder(OrderValue) startOrder(OrderKey) suspendOrder(OrderKey) resumeOrder(OrderKey) abortOrder(OrderKey) removeOrder(OrderKey) setOrder(OrderValue, boolean) tryCreateOrders(OrderValue[]) tryStartOrders(OrderKey[]) tryAbortOrders(OrderKey[]) tryRemoveOrders(OrderKey[]) trySetOrders(OrderValue[]) getOrder(OrderKey) getOrders(OrderKey[], String[]) getOrders(OrderValue[], String[])	queryManagedEntities(QueryValue, String[]) added createOrderByValue(OrderValue) startOrderByKey(OrderKey) suspendOrderByKey(OrderKey) resumeOrderByKey(OrderKey) abortOrderByKey(OrderKey) removeOrderByKey(OrderKey) setOrderByValue(OrderValue, boolean) tryCreateOrdersByValues(OrderValue[]) tryStartOrdersByKeys(OrderKey[]) tryAbortOrdersByKeys(OrderKey[]) tryRemoveOrdersByKeys(OrderKey[]) trySetOrdersByValues(OrderValue[], boolean) getOrderByKeyAllAttr(OrderKey) getOrdersByKeys(OrderKey[], String[]) getOrdersByTemplates(OrderValue[], String[])
javax.oss.order.OrderValue	-	In this implementation two attributes are optional: • PRIORITY • SERVICES
javax.oss.AttributeAccess	-	New: getSupportedOptionalAttributeNames() New exception • **getAttributeValue**(java.lang.String attributeName)throws IllegalArgumentException, IllegalStateException, UnsupportedAttributeException • public java.util.Map **getAttributeValue**s(java.lang.String[] attributeNames)throws IllegalArgumentException, IllegalStateException, UnsupportedAttributeException • public void **setAttributeValue**(java.lang.String attributeName, java.lang.Object Value)throws IllegalArgumentException, UnsupportedAttributeException • public void **setAttributeValues**(java.util.Map attributeNamesAndValuePairs) throws IllegalArgumentException, UnsupportedAttributeException

Table D-1 (Cont.) JVT API changes

-	ASAP 4.5	ASAP 4.6.x and later
javax.oss.ManagedEntityValue	-	Added setLastUpdateVersionNumber(long)
javax.oss.ManagedEntityKey	-	New methods • makeApplicationContext() • setApplicationContext(ApplicationContext) • setApplicationDN(String)
javax.oss.ManagedEntityKeyResult	-	New methods: • setException(Exception) • setManagedEntityKey(ManagedEntityKey) • setSuccess(Boolean)
javax.oss.ApplicationContext	-	Add the following methods: • clone() • setFactoryClass(String) • setSystemProperties(Map) • setURL(string)

Java provisioning API changes

The following table provides a comparison between the Java Provisioning APIs used in ASAP 4.5 and ASAP 4.6.x.

Table D-2 Java provisioning API changes

ASAP 4.5	ASAP 4.6.x and later
Packages: • com.nortel.pc • architel.jinterpreter	Packages: • com.mslv.activation • com.mslv.activation.jinterpreter
Configure connection handler in tbl_comm_param	Configure connection handler in tbl_nep_asdl_prog
Specify 'J' device type in tbl_comm_param	Specify interpreter type in tbl_nep_asdl_prog
Configure directly into database using SQL	Configure using XML and apply using JMX MBeans
Class Diagnosis deprecated	Class Diagnostic added
set/getExitType deprecated	set/getASDLExitType added
set/getExitType deprecated from class NEConnection	-