Oracle® Enterprise Manager

Application Developer's Guide

Release 1.6

June 1998

Part No. A63733-01

ORACLE®

Enabling the Information Age™

Oracle Enterprise Manager Application Developer's Guide

Part No. A63733-01

Release 1.6

Contributors: Robert Abbott, Eric Belden, Maureen Byrne, Dennis Lee, Yong-Chwen Liu, Dimitri Nakos, Bob Purvy, Jay Rossiter, Michael Stern, Dave Stowell

Contents

8 Event Management Integration

9 Jobs and Events Scripts

Send Us Your Comments

Oracle Enterprise Manager Application Developer's Guide Release 1.6

Part No. A63733-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways

- electronic mail - infosmp@us.oracle.com
- FAX - (650) 506-7200. Attn: Oracle System Management Products
- postal service
  Oracle Corporation
  Oracle System Management Products Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065 U.S.A.

If you would like a reply, please give your name, address, and telephone number below.

Preface

This preface describes the purpose and organization of the *Oracle Enterprise Manager Application Developer's Guide*. It also illustrates the conventions used in this guide. The preface contains the following information:

- Purpose of this Guide
- Knowledge Assumed of the Reader
- Code Examples
- How this Guide Is Organized
- Documentation Set
- Related Publications
- Conventions Used in This Guide
- Your Comments Are Welcome

Purpose of this Guide

This guide presents a general overview of developing applications that integrate into Oracle Enterprise Manager (OEMGR). It also discusses common services and using Tcl and OraTcl to write job and event scripts. This guide is written for software developers who wish to integrate their applications into Oracle Enterprise Manager.

Note: Information in this guide applies to Oracle Enterprise Manager running on the Windows NT 32 Bit platforms.

Knowledge Assumed of the Reader

This guide assumes you are familiar with the Oracle Enterprise Manager and the administration tasks that you can perform with Enterprise Manager components. If you are not, refer to the Oracle Enterprise Manager documentation set for a description of the tasks that you can perform with Enterprise Manager tools. In addition, this guide assumes you are familiar with the Oracle database server. For information about the Oracle database, see the Oracle server documentation set for specific and thorough descriptions of the database administration tasks and recommendations on how to administer your database optimally.

This guide also assumes that you are familiar with the operation of Microsoft Windows NT, Microsoft's Object Linking and Embedding (OLE) technology, the Microsoft Foundation Classes (MFC) and the Microsoft Visual C++ language. Please refer to the appropriate Microsoft documentation, if necessary.

Code Examples

Throughout this book there are references to source code files where you can find examples of the functionality that is being discussed. The code examples are included in the Enterprise Manager Software Developer's Kit (SDK) installation option that is available with the Oracle Enterprise Manager release. For information about installing Oracle Enterprise Manager components, see the Oracle Enterprise Manager installation and configuration documentation.

The installed code samples are installed in subdirectories of the `ORACLE_HOME\SYSMAN\SDK` directory. Microsoft Visual C++ makefiles have been included with the code, allowing you to build and debug the application yourself. You may need to make minor modifications to the makefiles to successfully build the sample application. See the SDK `readme` file in the `ORACLE_HOME\SYSMAN\SDK` directory

for information on how to modify the makefiles and install the application after it is built.

How this Guide Is Organized

This guide is divided into the following chapters.

Chapter 1, "Introduction"

This chapter describes the components of Oracle Enterprise Manager, the common services, and some programming considerations.

Chapter 2, "Tool Palette and Menu Integration"

This chapter describes how to register and launch an application from the Enterprise Manager Console.

Chapter 3, "General Coding Techniques"

This chapter describes initializing an application's document object, initializing dispatch drivers, and error handling.

Chapter 4, "Repository Control Integration"

This chapter describes how to store and retrieve Enterprise Manager repository information.

Chapter 5, "Navigator and Map Integration"

This chapter describes how to integrate with the Navigator and Map components.

Chapter 6, "Discovery Cache Integration"

This chapter describes how to retrieve information from the discovery cache.

Chapter 7, "Job Scheduling Integration"

This chapter describes how to integrate with the Job Scheduling system.

Chapter 8, "Event Management Integration"

This chapter describes how to integrate with the Event Management system.

Chapter 9, "Jobs and Events Scripts"

This chapter describes how to write job and event scripts using the Tcl language.

Appendix A, "NLS Codes"

This appendix lists the National Language Support codes.

Documentation Set

The Oracle Enterprise Manager product documentation includes the following:

- The *Oracle Enterprise Manager Readme* provides important notes regarding the updates to the software, documentation, and late-breaking news.
- The *Oracle Enterprise Manager Installation Guide* provides information about installing Oracle Enterprise Manager components. This guide is in an insert in the Enterprise Manager CDRom.
- The *Oracle Enterprise Manager Configuration Guide* provides information about configuring Oracle Enterprise Manager components.
- The *Oracle Enterprise Manager Concepts Guide* provides an overview of the Enterprise Manager system.
- The *Oracle Enterprise Manager Administrator's Guide* describes how to use the components and features of the Oracle Enterprise Manager system.
- The *Oracle Performance Monitoring User's Guide*, *Oracle Expert User's Guide*, *Oracle Trace Developer's Guide*, and *Oracle Trace User's Guide* provide information about performance monitoring applications.
- The *Oracle Enterprise Manager Messages Manual* describes the Oracle Enterprise Manager error messages and methods for diagnosing the messages.
- The *Oracle Enterprise Manager Application Developer's Guide* provides information on programming external interfaces to Oracle Enterprise Manager. The guide includes information on using Tcl and OraTcl to write custom job and event scripts.
- The *Oracle Enterprise Manager SNMP Support Reference Guide* describes the Oracle SNMP support feature and the public and private MIBs that support its use with various products.

The guides are available on the Oracle Enterprise Manager CD in HTML format for viewing with a web browser. In addition to the Enterprise Manager documentation, extensive online help is provided for Oracle Enterprise Manager components.

Related Publications

The *Oracle Enterprise Manager Administrator's Guide* refers to important information in the related publications. Depending on the version of the Oracle database, you would refer to the Oracle7 or Oracle8 documentation. The related books referred to in this guide are listed below:

- For general information about the Oracle Server and how it works, see the *Oracle Server Concepts Guide.*
- For information about administering the Oracle Server, see the *Oracle Server Administrator's Guide.*
- For information about administering Oracle Parallel Servers, see the *Oracle Parallel Server Support for the Oracle Enterprise Manager Console Guide.*
- For information about developing database applications within the Oracle Server, see the *Oracle Server Application Developer's Guide*.
- For the procedures on migrating from a previous version of Oracle, see the *Oracle Server Migration*.
- For information on Oracle's SQL commands and functions, see the *Oracle Server SQL Reference.*
- For information about Oracle's procedural language extension to SQL, PL/SQL, see the *PL/SQL User's Guide and Reference*.
- For information about Oracle messages and codes, refer to *Oracle Server Messages*.
- For information about the utilities bundled with the Oracle Server, including Export, Import, and SQL*Loader, refer to the *Oracle Server Utilities.*
- For information about distributing and replicating data, refer to the Oracle server distributed database systems and replication documentation.
- For information about the Oracle networking system for Oracle8, see the Net8 documentation, which includes the *Net8 Administrator's Guide*. For information about SQL*Net, see the SQL*Net documentation, which includes the *Oracle Network Manager Administrator's Guide*.
- For information specific to the Oracle Server working on your host operating system, see your operating system-specific Oracle documentation (specific book titles vary by operating system) and system release bulletins, if available.

Conventions Used in This Guide

This section explains the conventions used in this guide.

Special Words

Special words are provided to alert you to particular information within the body of this guide and within other manuals.

Note: Alerts you of important information.

Additional Information: References other guides and operating system-specific documentation.

Attention: Highlights information that is important to remember.

Suggestion: Signifies helpful suggestions and practical hints.

Warning: Indicates information that you should be aware of before you perform the action described.

Code Conventions

Code examples are shown in the following font:

```
command
```

When describing the syntax of code examples, arguments in square brackets '`[option1|option2]`' are optional. The '`|`' signifies the 'or' condition.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. Included in our manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address.

Documentation Manager
System Management Products
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065 U.S.A.

1

Introduction

This chapter introduces Oracle Enterprise Manager (OEMGR) and the process of integrating applications with Enterprise Manager. Topics include:

- Components of Oracle Enterprise Manager
- Console
- General Programming Considerations

Components of Oracle Enterprise Manager

Oracle Enterprise Manager (OEMGR) is Oracle's new generation of enterprise-wide system management tools that solves the problem of managing distributed Oracle systems with:

- A management Console that provides a single point-of-control.
- A Communication Daemon that provides communication between the Console and the intelligent agents.
- Intelligent agents that communicate with the Console to provide information about services located on the node where the agent is located.
- A set of common services.
- A set of integrated applications and utilities.

System management applications and utilities developed by Oracle or by third-party vendors can be integrated with the Console, providing a single point of control for system management.

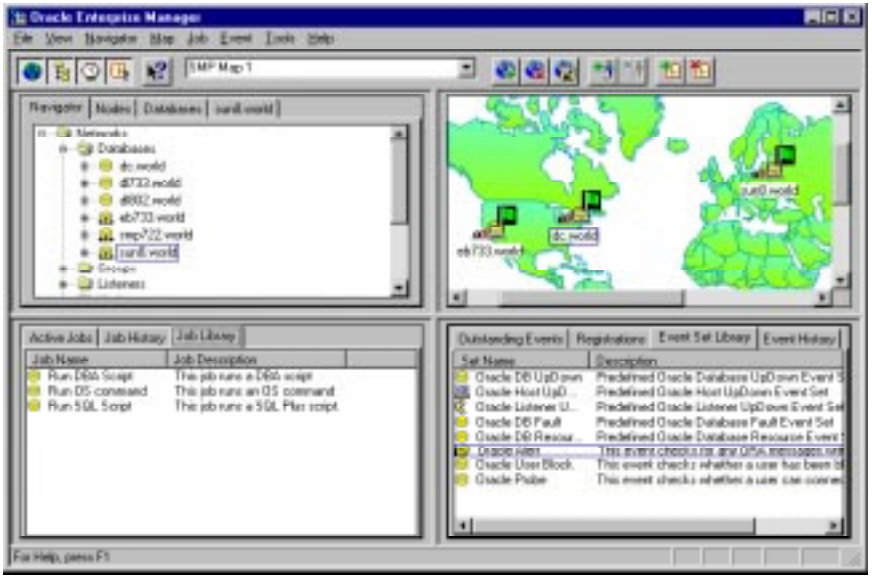
Object-Oriented Design

The Console and integrated applications communicate using Microsoft's OLE2 technology. This technology allows applications to easily integrate into the Console. This guide describes the set of application programmer interfaces (APIs) to Enterprise Manager Console and its services. These external interfaces allow developers to create tools that integrate into the Console. By adding applications that Oracle, third-parties, or you create, you can enhance or increase the scope of Oracle Enterprise Manager's capabilities.

Console

To the end user of Oracle Enterprise Manager, the Console looks like a window with several components. See Figure 1–1, "Oracle Enterprise Manager Console".

Figure 1–1 Oracle Enterprise Manager Console



The components that function with the Oracle Enterprise Manager Console are:

- Menus and tool palettes
- Repository
- Discovery Cache
- Navigator
- Map
- Job Scheduling
- Event Management
- Intelligent Agent

Menus and Tool Palettes

You can launch an application from the Console by clicking the application's icon in a tool palette or by selecting the application's menu item in the Tools menu. When you integrate your application, your application's icon appears in the appropriate tool palette, and a menu item for your application appears in the Tools menu.

You can integrate your application into Oracle Enterprise Manager Console with tool palette and menu integration. See Chapter 2, "Tool Palette and Menu Integration".

Console Services

Underlying the Console are various common services that applications can use. These services are:

- Repository
- Discovery Cache
- Navigator and Map - user interface (UI) only
- Job Scheduling System - UI and back-end
- Event Management System - UI and back-end
- Intelligent Agent

Repository

The repository is used to store system information that Oracle Enterprise Manager collects and uses. The repository consists of a set of internal tables installed in an Oracle database either locally on the host PC or remotely on the network. The repository external interfaces allow application developers to retrieve user preferences from the repository and to store their own information in the repository. See Chapter 4, "Repository Control Integration".

Discovery Cache

The discovery cache is responsible for servicing requests for information about the network and its services. The cache holds the current state of each node and service on the network. The discovery cache keeps track of:

- Nodes
- Listeners
- User-defined groups
- Oracle databases, including parallel instances
- External services (third-party services)

Information in the discovery cache is retrieved with the Navigator Service Discovery option or by reading Oracle Parallel Server information in the `topo_ops.ora` file. The discovery cache also keeps track of user-defined groups. A Console user can create named groups of nodes or services. A group can contain only nodes or services of the same type. For example, a user can create a group of listeners or a group of databases. A group can also contain a sub-group. The sub-group must be of the same type as the group. For example, a group of databases can also contain a sub-group of databases.

The discovery cache is also responsible for discovering and maintaining lists of all user-defined service types. User-defined services are accounted for in the same way as internally defined service types.

The discovery cache interfaces allow application developers to retrieve information about nodes, services, and user-defined groups. See Chapter 6, "Discovery Cache Integration".

Navigator and Map

The Navigator provides a tree listing of the nodes, objects, services, and other objects that can be administrated in the system. Each object type is identified by an icon. You specify the icons you want to appear for the services. You can administer each of these objects individually, and you can navigate between them by expanding and collapsing parts of the Navigator tree. The Map provides a graphical view of the objects in the system.

You can define your own service types to appear within the Navigator and on the Map. You must implement an interface that the discovery cache will call to discover which services are a part of the network environment. After these services have been discovered, they appear in the Navigator and the Map just as any internally-defined service type, such as databases or listeners.

You can use the Related Tools menu to launch applications to administer selected services. You can also use the Quick Edit command to launch property sheets from the Console user interface. From within the Console, you can specify preferred credentials for each of your services.

See Chapter 5, "Navigator and Map Integration".

Job Scheduling

The Job Scheduling system allows users to create, schedule, and delete jobs. The Console user interface provides property sheets and dialog boxes to perform these operations, and to view information about jobs in the system. The Job Scheduling system interfaces allow you to submit both batch and interactive jobs.

A job is a Tcl script that is executed remotely via the Oracle intelligent agent. There are predefined jobs that ship with Enterprise Manager, but applications developers can write new job scripts using the Tcl language. For information on job scripts, see Chapter 9, "Jobs and Events Scripts".

The Job Scheduling system external interfaces allow application developers to create and manage jobs. The Job Scheduling system can also notify the submitter of a job whenever there is a change in the job's status. See Chapter 7, "Job Scheduling Integration".

Event Management

The Event Management system (EMS) allows you to set up, register, and manage events on selected destinations in the network. EMS provides a mechanism for alerting administrators about possible or actual problems with selected services, such as databases or nodes. An event is a Tcl script that is executed remotely via the Oracle intelligent agent. There are numerous predefined events that ship with Enterprise Manager.

You can integrate into the EMS in two ways. On the Console machine, you can build an application which registers interest in a set of events and gets notifications when they are fired. On a remote node, you can send your own customized event information through the Oracle Intelligent Agent back to the Console machine. Integrating in both of these ways together can provide a powerful end-to-end communications from your remote services to your administration applications. See Chapter 8, "Event Management Integration".

General Programming Considerations

Some general programming considerations are:

Language Examples Use of OLE2 does not commit the third-party developer to the use of any particular programming language or model. However, throughout the documentation and code examples, Microsoft Foundation Classes (MFC) and Microsoft Visual C++ conventions are used. You do not have to use this class library or compiler, but it is strongly recommended.

Header and Library Files The SDK provides a library of C++ classes and functions which make the development of integrated applications much easier. Classes are provided to save you the trouble of implementing much of the routine, uninteresting, or complicated OLE2 process.

All of the code you need to build using the library is provided within the `SYSMAN\SDK\VOX` subdirectory in the `ORACLE_HOME` directory.

Note: The `ORACLE_HOME` directory is the location where Oracle products have been installed. The default for Windows NT is the `ORANT` directory.

Two static import libraries are provided within the `SYSMAN\SDK\VOX\LIB` subdirectory: `vox.lib` for release builds, and `voxd.lib` for debug builds. These correspond to the two DLLs that your applications will use, `vox.dll` and `voxd.dll`, both of which are supplied with the Oracle Enterprise Manager Console. Note that you can link with and run the debug DLL with a debug version of your application yet still execute side-by-side with the release (non-debug) version of the console. All of the header files that you might need to include can be found in the `ORACLE_HOME\SYSMAN\SDK\VOX\INCLUDE` subdirectory.

Sample Applications The SDK comes with two sample applications that provide examples of calls to the Oracle Enterprise Manager Console OLE APIs and demonstrate the use of the VOX library. The first application, `smpsrv` (SMP sample SeRVer), demonstrates the use of almost all the functionality exposed by the console, including launch in context, repository and discovery cache calls, job submission and notification, and event registration and notification. The second application, `smpxsrvc` (SMP eXternal SeRViCe), is an example of how to create an OLE service for Navigator and Map integration. All of the files necessary to each project can be found within the `SYSMAN\SDK\` subdirectories.

Makefiles are included with the samples in order to build working executables. The makefiles are configured based on the assumption that your `ORACLE_HOME` is `C:\ORANT`. If it is not, you will need to reconfigure them accordingly. Move the

executables into the ORACLE_HOME\BIN subdirectory after they have been built, and then execute either smpsrv.reg or smpxsrvc.reg as appropriate. You then have working examples that you can use and debug. Most likely you will find it very useful during the development of your application to test its functionality against that of the smpsrv application.

Both of the sample applications are designed to be used primarily as examples of code use and integration techniques. They have not been developed as full fledged applications for end user use; their user interfaces are functional but unrefined. Most of the dialogs in smpsrv, for example, are just thin wrappers around the underlying API they are meant to demonstrate. They do not pretend to maintain state or re-synchronize properly, and as such should be used for one invocation of the API and then dismissed.

Throughout this book there are references to the source code files where you can find examples for the functionality that is being discussed.

Wide Characters All strings that are returned to your application are buffers of wide characters (UNICODE).

2

Tool Palette and Menu Integration

This chapter covers the Tool palette and menu interface. It describes:

- Integrating an Application into the Console
- Registering an Application with the Console
- Credential Precedence
- Passing Contextual Information to an OLE Automation Server Application
- Passing Contextual Information to a non-OLE Automation Server Application
- Defining a Custom Tool Palette
- API Reference

Integrating an Application into the Console

The Oracle Enterprise Manager Console allows you to register and launch an application directly from a tool palette or a pull-down menu. If you have registered an application in the NT registry, the Console uses registration information to make an application available to Console users. The Console can:

- Launch an application from a palette or menu.
- Launch an application and pass it connection details from the selected service or node.

In order for Oracle Enterprise Manager to know about an application, you must register it in the NT registry. When the Console starts up, it reads the registry and looks for applications that are to be made available in the Console.

- To have the Console simply launch an application, you need to register an application in the NT registry. See *Registering an Application with the Console* on page 2-2.
- To have the Console pass an application connection details, you must either implement a `SetLogonInfoEx` OLE automation interface or process the command line parameters that Oracle Enterprise Manager passes if you are using a non-OLE automation server. See *Passing Contextual Information to an OLE Automation Server Application* on page 2-8 or *Passing Contextual Information to a non-OLE Automation Server Application* on page 2-8.

Registering an Application with the Console

To register an application with the Enterprise Manager Console, entries must be made into the NT registry. Each application must define a unique *application key* under the `HKEY_CLASSES_ROOT\OracleSmpConsole\Applications` key. To avoid naming collisions with applications developed by other companies, you should begin the name of all of the application keys with a substring uniquely identifying your company. All of the details that the Console needs to know about the application should be placed underneath its application key. The syntax for an application's NT registration keys is:

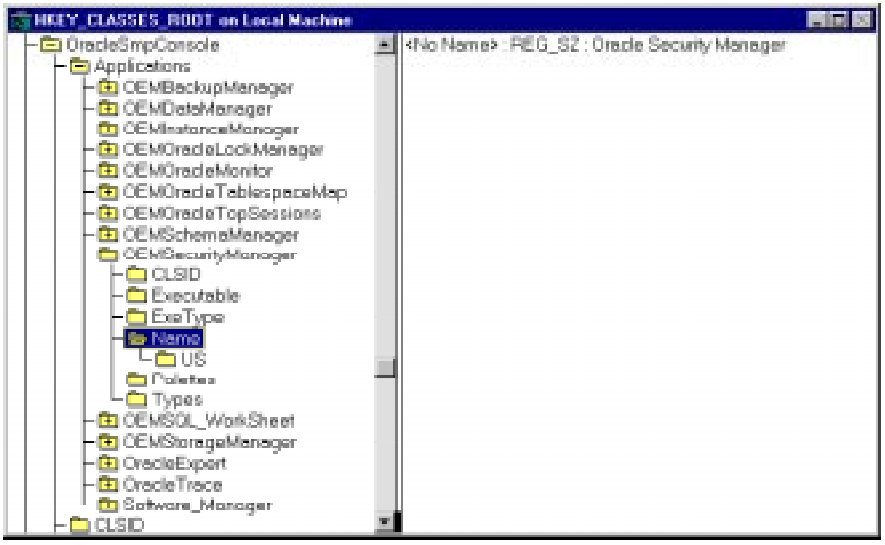
`HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\AppNameKey\r`*`egkey = regkeyvalue`*
where:

- `OracleSmpConsole\Applications` identifies the application as one that should be displayed in the Console. This is the specific location where the Enterprise Manager Console looks for all registered applications.

- `AppNameKey`, such as `Smpsrv`, is the key that is used by the NT Registry to organize the subkeys that relate to the application. This key is also used by the Console internally to refer to an application.
- *`regkey`*, such as ExeType, is one of registration keys.
- *`regkeyvalue`*, such as OLE_AUTOMATION, is a registration key value.

Figure 2–1, "Example of an NT Registry" shows a sample NT registry. It includes an entry for the `SmpSrv` test application registered with the Console. For a complete example of the keys used to register an application in the registry, see the `smpsrv.reg` file.

Figure 2–1 Example of an NT Registry



The Console uses an application's registration information to:

- Gather details relating to how the application should be launched.
- Determine the application's National Language Support (NLS) name.
- Display an application's icon in the appropriate tool palette.
- Include a menu item for an application in the appropriate submenu of the Tool menu.

Required Registration Keys

To register an application with the Enterprise Manager, you must specify the following four keys:

1. The application key: The internal name for the application.
2. Name: The readable application name key, with NLS subkeys.
3. Executable: The executable filename.
4. ExeType: The executable type.

Application Key

The value of the application key is the string that is used internally by the Console to identify the application. Both the application key and value must be unique. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample = OEMSample
```

Note: Make sure that there are no spaces trailing the application key string because these spaces are considered part of the key value.

Readable Name

This is the name displayed by the Console in the Tools menu. The name is also displayed in bubble help and status bar help for the palette. All readable strings that Oracle Enterprise Manager requires to be specified in the registry, including the application names, handle national language translation in the same way. The `Name` key's value is the default name for the application, which should be in American English. The console will use this string if it is running in a language for which no NLS subkey has been specified. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Name =
OEMSample Application
```

For every language the application can support, an NLS subkey, or language code, should be specified. For example, the Japanese name would include JA instead of US in the registration key syntax. For a list of the subkeys, see Appendix A, "NLS Codes". The value of a subkey is the name of the application in the appropriate language. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Name\US =
OEMSample Application
```

Executable Filename

The executable name of the application. The executable itself must be installed in either your `ORACLE_HOME\BIN` subdirectory or within the current path. If the executable is located anywhere else, the application will fail to appear in the tool palettes and menus you desire. The executable file name should be supplied without path information. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Executable =
SMPSRV.EXE
```

Executable Type

The type of executable file. The executable type can be one of the values listed in Table 2–1, "Executable Types".

Table 2–1 Executable Types

Value	Meaning	Comments
OLE_AUTOMATION	OLE Server	Overwhelmingly recommended. Must implement SetLogonInfoEx interface
EXEC	Simple executable	Simple launch without context
EXEC_PARAMS	Command line launch-in-context	Same context info as in SetLogonInfoEx, but passed on the command line

For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\ExeType =
OLE_AUTOMATION
```

Additional steps that are necessary to launch each of the executable types are discussed in following sections of this chapter.

Optional Registration Keys

There are several optional keys that each application can specify: the `Palettes`, `Types`, `CLSID`, and `Icon` keys. Oracle recommends that you specify `Palettes` and `Type`. `CLSID` is required to register an OLE Automation server application.

Palettes

This key determines the list of palettes where the application is displayed, usually only one. Multiple entries are separated by commas. Use of the `Palettes` key is optional but strongly recommended. Define this key to specify on which Tools Palette the application will appear within the console.

If you list a particular Tools Palette, the application's icon will appear, and its (NLS) name will appear in Tooltips and status bar help when the mouse lingers over that icon for a brief period of time. In addition, there will be a corresponding menu item included in the appropriate location in the Tools menu.

Tools palettes are specified by their internal names. Multiple tools palettes can be specified: the value of the "Palettes" key should be a comma separated list. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Palettes =
OracleApps,SampleApps
```

There is a default tool palette whose internal name is `OracleApps`. Its external name is simply `Applications`, which is generic enough that most applications can be put there. It is also possible to define your own tool palette. See *Defining a Custom Tool Palette* on page 2-10.

Types

This key determines the types of objects that this application administers. If you specify the `Types` key, the application will appear in the Related Tools contextual menu for all the types you list in the key's value. Types are specified by their internal type name. Multiple entries are separated by commas. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Types =
DATABASE,NODE,LISTENER
```

Types specified can be those defined by Oracle Enterprise Manager itself as they appear in the header file `voxtype.h`, or can be those which you or another third-party vender define as external service types. See Chapter 5, "Navigator and Map Integration".

Related Tools menus pop up when you click the right mouse button on an object in the Navigator. A list of applications which are appropriate to the administration of objects of that type appear as a submenu. Selection of one of these applications launches it with the context of the selected object.

CLSID

To register an OLE Automation server application, one additional registry key is required, the `CLSID` key. Specify the CLSID of the OLE server as the value of this key. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\CLSID =
{FC2B0F50-0036-11CF-BEF1-0020AF0C14D1}
```

The server must implement and expose the `SetLogonInfoEx` interface to receive launch-in-context information.

Icon

To change the icon that an application uses in the tool palette, use the `Icon` key. The replacement icon must be stored in bitmap format (*.bmp). For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\Icon =
%oracle_home%\sysman\admin\vaxtpdus.bmp
```

If an icon is not found for an application, a default icon (`oracle_home\sysman\admin\vaxtpdus.bmp`) is automatically used. This default icon is also used if an integrated application is not an MFC application and its executable does not contain an icon. The `tbicon.reg` file in the `oracle_home\sysman\sdk` directory currently points to the default icon.

Credential Precedence

Part of the context of an application's launch is the logon credentials for the service that is associated with the selected object in the Navigator tree or on the Map. These credentials include the username and password, plus role if the object selected is a database service. The role specifies how to connect to the database: NORMAL, SYSOPER, or SYSDBA. The Console allows an individual user to specify preferred credentials for any service, including those defined externally by third parties. For information, see Chapter 5, "Navigator and Map Integration".

Also, at any given time it may be possible to be connected to a service within the console under arbitrary credentials entered in a Login Information dialog. Accordingly, the precedence of the credentials which will be sent as context is determined according to the following rules:

1. If the user has already explicitly connected to the service, the arbitrary credentials entered in a Login Information dialog or the preferred credentials at the time of connection are sent.
2. If a user has not entered arbitrary credentials and has defined a preferred credential for the service, these are sent.
3. If a user has not entered arbitrary credentials and has not defined a preferred credential for the service, the current console logon credentials are sent.

Passing Contextual Information to an OLE Automation Server Application

When the Console launches an OLE automation application, it calls the application's `SetLogonInfoEx` automation interface, if such an interface has been defined in the application. See *SetLogonInfoEx* on page 2-11.

Note: If an application is a non-OLE automation server, the Console can still pass logon information if you have specified the `ExecType = EXEC_PARAMS`.

The Console passes the following connection information to the application:

- Credentials of the Console user for the node or service selected in the navigator.
- Object name and type selected in the Navigator or Map.

OLE Launching Considerations

The application must be an OLE server and must be able to launch with the OLE `Open` verb. If the application is going to support external service types described in Chapter 5, "Navigator and Map Integration", it should also be able to launch with the `CreateDispatch` method.

Passing Contextual Information to a non-OLE Automation Server Application

If the `ExeType` for an application is specified to be `EXEC_PARAMS`, contextual information about the currently selected object in the Navigator tree or on the Map will be passed to the application via the command line. An example of specifying `EXEC_PARMS` is:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Applications\OEMSample\ExeType =
EXEC_PARMS
```

Seven named parameters separated by spaces are passed on the command line according to the following syntax:

```
paramter_name=value
```

The meanings of each of these parameters are described in Table 2–2, "Command-Line Parameters".

Table 2–2 Command-Line Parameters

Parameter name	Meaning	Notes
user	Username of service related to selected object	Defaults according to credential precedence rules.
password	Password of service related to selected object	Defaults according to credential precedence rules.
service	Name of service related to selected object	For example: If the selected object is a tablespace, the related service would be the database that tablespace is a part of.
type	Type of service related to selected object	Internal service type name, either from voxtype.h, or defined externally.
as	Connect as role, for databases only.	Only values are SYSOPER or SYSDBA. If NORMAL or service is not a database, this parameter is not supplied.
object	Name of selected object	
objecttype	Type of selected object	Internal type name, either from voxtype.h, or defined externally.

Defining a Custom Tool Palette

You can define custom tool palette to put applications on. However, because tool palettes look awkward if they have less than three applications on them, you should do this only if you have three or more applications in an application suite which you choose to integrate. If you have less than three applications to place on a palette, consider using the OracleApps palette. For information on specifying a palette, see *Optional Registration Keys* on page 2-5.

To create a custom palette, you only need to specify an internal name for the new palette and the human readable name, with NLS subkeys. The internal name is the keyword which will appear in the `Palettes` key for each application you want to place on your palette. The human readable name will be used in the title bar of the undocked palette, and will also be used to create a new entry under the Tools menu in the console to correspond to the new palette.

Registering a Custom Palette

To create a new tool palette that will appear in the Console, you need to add an *internal name*, *external name*, and *NLSkey* to the NT Registry. These entries are added under the `HKEY_CLASSES_ROOT\OracleSmpConsole\Palettes` key.

Internal Name

Internal palette name.The *name* must be a unique subkey to represent the palette you are registering. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Palettes\SampleApps = SampleApps
```

External Name

Default value for human readable name (US English). For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Palettes\SampleApps\Name =
Sample Applications
```

NLSkey

NLS value for human readable name. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Palettes\SampleApps\Name\US =
Sample Applications
```

After you have included the palette in the registry, you can add an application to it as described *Registering an Application with the Console* on page 2-2.

API Reference

In order for an OLE application to receive connection information from the Console, the application must expose a `SetLogonInfoEx` method.

Note: If you do not expose a `SetLogonInfoEx` method an application will still launch, but no credentials will be passed.

SetLogonInfoEx

Purpose `SetLogonInfoEx` allows the Console to pass the Console user's credentials and the selected object name to an application. It should have the following syntax:

Syntax
```
VT_EMPTY SetLogonInfoEx(VTS_BSTR Username,
                        VTS_BSTR Password,
                        VTS_BSTR Service,
                        VTS_BSTR ServiceType,
                        VTS_BSTR ConnectAs,
                        VTS_BSTR FocusedObectName,
                        VTS_BSTR FocusedObjectType)
```

Parameters

Name	Type	Mode
Username	VTS_BSTR	IN
Password	VTS_BSTR	IN
Service	VTS_BSTR	IN
ServiceType	VTS_BSTR	IN
ConnectAs	VTS_BSTR	IN
FocusedObjectName	VTS_BSTR	IN
FocusedObjectType	VTS_BSTR	IN

Username
Username for the service that defines an application's launching context.

Password
Password for the service that defines an application's launching context.

Service
Name of the service or node that defines the context an application is being launched in by the Console.

ServiceType
Type of service that defines the context an application is being launched in by the Console. For information on type parameters, see *Discovery Cache API Reference* on page 6-5.

ConnectAs
Role to be used when connecting to the database. Values are `NORMAL`, `SYSOPER`, or `SYSDBA`. The value is `NORMAL` if the ServiceType is not a database.

StrFocusedObject
Name of the focused object in the Navigator or Map.

TypeFocusedObject
Type of the focused object in the Navigator or Map.

Comments Enterprise Manager calls `SetLogonInfoEx` function to pass contextual information about the service. After the application has obtained this information from the Console, it can use that information to open a connection to a service.

If the current selection in the Navigator is a container, the `FocusedObjName` parameter will be an empty string and the `FocusedObjType` parameter will contain the type for that container. If the container appears under a service in the tree, that service will appear in the `Service` and `Type` fields, and the `Username`, `Password`, and `ConnectAs` credentials will be those which are in force for that service as determined by the credential precedence scheme discussed above. If the container does not appear under a service, the `Service` name will be the empty string, the `Type` field will be `VOXTYPE_TYPE_INVALID`, and the credentials will be the console logon credentials.

3

General Coding Techniques

This chapter covers the general coding techniques for writing an application. Topics include:

- Console Service Objects
- Initializing and Disposing Dispatch Drivers
- Retrieving Error Information
- VoxErrorUnpacker Class and Methods

Console Service Objects

The Oracle Enterprise Manager Console contains an OLE2 Automation Service object which exposes all of its external APIs, except those pertaining to job submission. The `ProgID` for this service is `OracleSmpConsole`. The Console starts up this service upon initialization, and then declares it to be the active object of its OLE class. As there can only be one console running at a time, it is guaranteed that there will be one and only one object of this class. You should use the MFC call `GetActiveObject` to retrieve a dispatch interface for the `OracleSmpConsole` object. You can then use this interface to call any of the automation methods described in this guide. For an example of the use of `GetActiveObject`, see `CSmpSrvDlg::GetDispatchDriver` in the `smpsrvdl.cpp` file.

If you are integrating tightly into the console; which means most of the application's functionality requires the console's back-end services; then you will want to try to connect to the `OracleSmpConsole` service when the application initializes. If this connection fails, you should display a message informing the user that the application requires the Oracle Enterprise Manager to be running and then abort the application.

In order for the Enterprise Manager Console to use the various interfaces that an application exposes, you must initialize an OLE automation server object, usually the application's document object, and register it as the active object. To do this:

1. Get the document's `IUnknown`.
2. Find out the class ID (`CLSID`) from the document object's ProgID.
3. Register this object as the active one.

For an example illustrating how to initialize and register the `Smpsrv` document object, see `CSmpsrvDoc::CSmpsrvDoc` in the `smpsrdoc.cpp` file.

Initializing and Disposing Dispatch Drivers

The following steps allow an application to call Enterprise Manager's external interfaces:

1. Use the service's `ProgID` to get the class ID (`CLSID`).
2. Ask `IUnknown` to get the active object of that class.
3. Call `QueryInterface` to get the `IDispatch` of the service.
4. Attach the `LPDISPATCH` to a `COleDispatchDriver` and return the dispatch driver.

For an example illustrating how to create and initialize a dispatch driver to use in calling the Oracle Enterprise Manager external interfaces, see `CSmpSrvDlg::GetDispatchDriver` in the `smpsrvdl.cpp` file.

When you are done with a dispatch driver you should dispose of it. For an example illustrating how to how to dispose a dispatch driver, see `CSmpSrvDlg::ReleaseDispatchDriver` in the `smpsrvdl.cpp` file.

The Job Object

Every time you want to submit a new job or delete an existing one, you must create a new OLE automation object of the class `OracleSmpJob`. An easy way to do this is to use the MFC call `CreateDispatch`. You then initialize the job and perform whatever manipulations you desire. The `OracleSmpJob` object gets destroyed when you are done. This method differs fundamentally from that of using any of the other APIs, where you call `GetActiveObject` on the `OracleSmpConsole` service which always exists and runs with the console.

For an example illustrating how to use `CreateDispatch`, see `CSmpSrvDlg::GetJobObject` in the `smpsrvdl.cpp` file.

Retrieving Error Information

Most of the functions in the Enterprise Manager external interfaces return a VT_BOOL.

- A TRUE value indicates that the function has succeeded.
- A FALSE value indicates failure.

Whenever an API fails, call GetErrorInfo to retrieve more information about the failure. Every OLE automation object in the Enterprise Manager Console implements its own GetErrorInfo interface.

GetErrorInfo

Purpose GetErrorInfo retrieves error information about the API which was executed last for a particular OLE service.

Syntax VT_VARIANT GetErrorInfo()

Comments This function returns a VARIANT that contains the specific error information for the interface function that failed. Use the helper functions described in the section *VoxErrorUnpacker Class and Methods* on page -5 to unpack the error information.

Note: You may call GetErrorInfo even if the function succeeded.

For an example illustrating how to use GetErrorInfo to retrieve error information and use the VoxErrorUnpacker class to unpack that information, see CSmpSrvDlg::DoGetErrorInfo in the smpsrvdl.cpp file. The dispatch driver varies according to the OLE service you want to request the error information for.

VoxErrorUnpacker Class and Methods

The class `VoxErrorUnpacker` has several methods which help you to unpack the error information from the `VARIANT` returned by `GetErrorInfo`.

- GetErrorText
- GetErrorCause
- GetErrorData
- Success
- Failure

Note: The class definition for `VoxErrorUnpacker` is in the file `voxerr.h`.

GetErrorText

Purpose Call the `GetErrorText` method to extract the text from the VARIANT returned by `GetErrorInfo`.

Syntax `const CString& GetErrorText()`

Comments `GetErrorText` returns a string which contains the error text.

GetErrorCause

Purpose Call the `GetErrorCause` method to extract the error number from the VARIANT returned by `GetErrorInfo`. The semantics of these numbers will be dependent upon the function that failed.

Syntax `long GetErrorCause()`

Comments `GetErrorCause` returns a LONG number which represents a specific error.

GetErrorData

Purpose Call the `GetErrorData` method to get other error data from the VARIANT returned by `GetErrorInfo`.

Syntax `const VARIANT& GetErrorData()`

Comments `GetDataError` returns a VARIANT that contains any other kind of data which was passed back. This data will depend on which function failed and what caused the error.

Note: Not all errors have associated data.

Success

Purpose Call the `Success` method to determine if an API function succeeded or not.

Syntax `BOOL Success()`

Comments `Success` returns a BOOLEAN that determines whether the called API function succeeded: TRUE for success and FALSE for failure.

Failure

Purpose Call the `Failure` method to determine if an API function failed or not.

Syntax `BOOL Failure()`

Comments `Failure` returns a BOOLEAN that determines whether the called API function failed: TRUE for failure and FALSE for success.

4

Repository Control Integration

This chapter covers the repository control interface. It describes:

- Accessing the Repository
- Getting Preferred Credentials
- Repository API Reference

Accessing the Repository

The repository control interface allows you to:

- Get the repository logon credentials so that you can store and access information specific to an application.
- Retrieve information about a user's preferred credentials.

If an application needs to access the repository, you need to use the repository control interface to establish a connection. If you want, you can store information specific to an application in the repository. Before you do this, you need to create the tables that an application will use. This can be done in a SQL script run during installation of the application. To store or access information in the repository:

1. Connect to the repository object. See *Initializing and Disposing Dispatch Drivers* on page 3-2.
2. Request connection information from the Console.
3. Make a connection to the repository.

Enterprise Manager maintains its own tables within the repository strictly for its private use. However, you may create your own tables within the repository to act as a persistent store for your application data. The advantages to doing this are:

- Use of the repository's security model to keep application data distinct. As this mirrors the security model used by the Console, the application appears to be more tightly integrated into the Enterprise Manager application suite.
- Centralized storage which can be accessed by the application's users from multiple console and application installations.
- When developing multiple applications, data in the repository can be shared.

Request Connection Information from the Console

Call `GetRepLogonInfo` to retrieve repository connection information from the Console. For an example of retrieving the information from the repository object, see the `CRepLogonDlg::DoGetRepLogonInfo` in the `replogon.cpp` file.

Connect to the Repository

After you have received the repository connect information from the Console, you can connect to the repository using the `olog` call of the Oracle Call Interface (OCI). When you have connected to the repository, you can execute any SQL or PL/SQL statements on the application's repository structures, such as tables or views.

Getting Preferred Credentials

Your application can also request a user's preferred credentials from the repository by calling `GetPreferredCredentials`. To retrieve the user's preferred credentials for a particular service you must do the following:

1. Connect to the repository object.
2. Request the user's preferred credentials for a particular service.

Request Preferred Credentials

Call `GetPreferredCredentials` to retrieve the preferred credentials for the indicated service. For an example illustrating how to retrieve the preferred credentials, see `CGetPrefCredDlg::DoGetPreferredCredentials` in the `reppref c.cpp` file.

Repository API Reference

This section describes the external interface calls for the repository service.

- GetConsoleVerison
- GetPreferredCredentials
- GetRepLogonInfo

Common Parameters

These parameters are used in multiple API calls and are described in this section.

pConnectAs
The role that is used to connect to the database: SYSOPER, SYSDBA, or NORMAL. If the `DestinationType` is not a database, `pConnectAs` is NORMAL.

pUsername
The username of the user.

pPassword
The password of the user.

GetConsoleVerison

Purpose This function returns the release version for the Console.

Syntax
```
VT_BOOL GetConsoleVersion(VTS_PBSTR pVersion,
                          VTS_PBSTR pBanner)
```

Parameters

Name	Type	Mode
pVersion	VTS_PBSTR	OUT
pBanner	VTS_PBSTR	OUT

pVersion
This identifies the console version, such as *1.2*. It is used when programing.

pBanner
This is the NLS version banner string for display in the user interface (UI), such as the About box.

Comments For an example illustrating `GetConsoleVersion`, see `CGetVersionDlg::DoGetConsoleVersion` in the `repver.cpp` file.

GetPreferredCredentials

Purpose This function returns the preferred credentials for the Console user for a particular destination. If the user has not specifically set credentials for the destination, the Console username and password are returned.

Syntax
```
VT_BOOL GetPreferredCredentials(VTS_BSTR DestinationType,
                VTS_BSTR Destination,
                VTS_PBSTR pUsername,
                VTS_PBSTR pPassword
                VTS_PBSTR pConnectAs)
```

Parameters See *Common Parameters* on page 4-3.

Name	Type	Mode
DestinationType	VTS_BSTR	IN
Destination	VTS_BSTR	IN
pUsername	VTS_PBSTR	OUT
pPassword	VTS_PBSTR	OUT
pConnectAs	VTS_PBSTR	OUT

DestinationType
The service type of the destination. See *Service Types* on page 6-3.

Destination
The name of the destination, such as *smpsun14*.

Comments An application would call the `GetPreferredCredentials` function when it wants to open a connection to a different service than the one it connected to when it was launched. This function returns a TRUE if it succeeds, otherwise it returns FALSE.

GetRepLogonInfo

Purpose This function returns the username and password of the Console user, and the service name of the repository being used by the Console user.

Syntax
```
VT_BOOL GetRepLogonInfo(VTS_PBSTR pUsername,
                        VTS_PBSTR pPassword,
                        VTS_PBSTR pService
                        VTS_PBSTR pConnectAs)
```

Parameters See *Common Parameters* on page 4-3.

Name	Type	Mode
pUsername	VTS_PBSTR	OUT
pPassword	VTS_PBSTR	OUT
pService	VTS_PBSTR	OUT
pConnectAs	VTS_PBSTR	OUT

pService
The Service name of the repository.

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE.

5

Navigator and Map Integration

This chapter covers the integration with the Navigator and Map. Topics include:

- Integration of Services
- Registering Service Types
- Integrating Application's APIs

Integration of Services

Discovery takes place when the Console boots up, and can occur at other times. The discovery mechanism is determined by the integrating application. Once in the Discovery Cache, a type is available for use in the Navigator, Map, Job, and Event interface. Anywhere there is a list of types, a new type will appear along with a list of services of the new type with the appropriate icons.

The most direct consequence of defining an external service type is integration into the Navigator and Map within the console user interface. The implementation provides the necessary bitmaps and icons for display of the services. Just as for internally-defined services, objects of external service types can be dragged and dropped onto the map. When the `QuickEdit` command is invoked from either the Navigator or the Map, an executable is called to launch a property sheet for the administration of the selected object. If the Related Tools command is selected from the Navigator, the user can choose to launch any applications which have been declared for the administration of the service types. Those applications will be passed details about the selected service, including name, type, and credential information. For more information, see Chapter 3, "General Coding Techniques".

In addition, you can schedule jobs to services of types that you have defined. During service discovery, the implementation specifies not only the name of your service, but also the name of the node on which the service appears. Fundamentally, the running of a job is the execution of Tcl script by an Oracle Intelligent Agent on a particular node. For example, a database job only differs from a node job by description of their effects. If the nodes that you associate with your services have agents installed upon them, you can schedule jobs to those services. For additional information, see Chapter 6, "Discovery Cache Integration".

The OLE service within which you define your service types is created via a call to the `COleDispatchDriver::CreateDispatch` API based on the `CLSID` that you provide in the NT registry. The OLE service gets re-created for each call to any of its interfaces.

One OLE application service can support multiple, externally-defined service types using the same set of APIs. Because of this, all the API's take `Type` as the first parameter. Externally-defined services get stored in the discovery cache the same as internally-defined types. They stay discovered until the services are explicitly deleted from the Console.

Notes on External Services

When defining an external service in the Navigator tree, the following limitations apply to this release:

- The service objects can only be displayed in the top-level container defined for the service types.
- The object cannot appear in or have subtrees.

Registering Service Types

For each service type you want to add, enter a unique subkey into the NT registry under `HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects`. You must provide values for the internal and external name keys, and `CLSID`. See the `smpxsrvc.reg` file for a complete example of a registration file.

Internal Type

This is the name of the type as it is represented internally within the discovery cache. It also serves as the string which represents the type for all of the APIs within the SDK. This internal type name must be unique. Compare it to other types under `HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects` and to those defined in the `voxtype.h` header to ensure that this is the case. Oracle recommends that you use the company name as a prefix. You supply the internal type name as a value for the subkey which represents your type. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects\Apple = APPLE
```

External (NLS) Type

This is the name string that Enterprise Manager uses in its user interface to list the type among others. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects\Apple\Name = Apples
HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects\Apple\Name\US = Apples
```

See *Integrating an Application into the Console* on page 2-2 for information on registering an application in the NT registry. See Chapter 2, "Tool Palette and Menu Integration" for the method of defaulting and translation of this string.

CLSID

This provides the Enterprise Manager Console with the `CLSID` of the OLE service that defines the type. For example:

```
HKEY_CLASSES_ROOT\OracleSmpConsole\Navigator\Objects\Apple\CLSID =
{46CD44FC-C541-11CF-8F38-0020AFF2B3FF}
```

Integrating Application's APIs

An application should expose the following APIs:

- Discover
- GetIconList
- GetDefaultDisplayInfo
- QuickEdit

Common Parameters

This parameter is used by multiple Integrating APIs and is described in this section.

Type
The internal name for that type as it appears in the registry. See *Optional Registration Keys* on page 2-5.

Discover

Purpose The Console calls this when it needs to discover the list of services, which will be stored in the Discovery Cache. For services that can be discovered by the agent, see *Discovering Services* on page 6-4.

Syntax `VTS_BOOL Discover(VTS_BSTR Type, VTS_PVARIANT pServices)`

Parameters See *Common Parameters* on page 5-4.

Name	Type	Mode
Type	VTS_BSTR	IN
pServices	VTS_PVARIANT	OUT

pServices
A list of the names of the services of a specified type. pServices is an array of the form:

```
{ { Name1, Node1 }
  { Name2, Node2 }
  ...
  { Namex, Nodex } }
```

where *Namex* is the service name. *Nodex* is the name of the node the service is located on. This is used for job submission and event registration.

Comments One OLE service can discover multiple types over multiple calls.

GetIconList

Purpose Used by the Console to get the icons it needs to use in the Navigator and map when displaying your services. This function is obsolete in Enterprise Manager release 1.3.5 and later releases. See *GetDefaultDisplayInfo* on page 5-6.

Syntax `VT_BOOL GetIconList(VTS_BSTR Type, VTS_PVARIANT pIconOffsets)`

Parameters See *Common Parameters* on page 5-4.

Name	Type	Mode
Type	VTS_BSTR	IN
pIconOffsets	VTS_PVARIANT	OUT

pIconOffsets
An array of offsets that reference which icons to use for the specified type. pIconOffsets contains exactly four VTS_I4 which represent, in order:

1. Offset of the tree icon (13x13)
2. Offset of the tree group icon (13x13) [not yet used]
3. Offset of the map icon (32x32)
4. Offset of the map group icon (32x32)

You must always pass a 32x32 icon, but for tree bitmaps, the Console only uses the upper left hand 13x13 portion of the icon. The unused area for small bitmaps (13x13) is ignored. A large icon is 32x32.

Comments These offsets correspond to the order in which the icons are listed in the .exe file. These are the same as the offsets you would use in a call to the Windows SDK ::ExtractIcon API.

GetDefaultDisplayInfo

Purpose Used by the Console to get the icons it needs to use in the Navigator and map when displaying your services.

Syntax
```
VT_BOOL GetDefaultDisplayInfo(VTS_BSTR Type,
                VTS_PBSTR pbstrDisplayName
                VTS_PBSTR pbstrIcon)
```

Parameters See *Common Parameters* on page 5-4.

Name	Type	Mode
Type	VTS_BSTR	IN
pbstrDisplayName	VTS_PBSTR	OUT
pbstrIcon	VTS_PBSTR	OUT

pbstrDisplayName
This is the NLS display name for this type.

pbstrIcon
This is the string listing of the icons to use for this type. Four numeric values are expected. These values correspond to the resource Ids (in the `.h` resource file) for the icons. The offset are, in order:

1. Offset of the tree icon (13x13)
2. Offset of the tree group icon (13x13) [not yet used]
3. Offset of the map icon (32x32)
4. Offset of the map group icon (32x32)

Comments This function returns `TRUE` if the `Type` was handled successfully.

QuickEdit

Purpose This is called to launch a property sheet to administer an object `ObjectName` of type `Type`. It is called when the Quick Edit menu item is selected for an object of the selected type in the Navigator. This property sheet should remain under the OLE server's control until the user explicitly closes it.

Syntax `VT_BOOL QuickEdit(VTS_BSTR Type, VTS_BSTR ObjectName)`

Parameters See *Common Parameters* on page 5-4.

Name	Type	Mode
Type	VTS_BSTR	IN
ObjectName	VTS_BSTR	IN

ObjectName
The name of the service object.

Comments It is expected that your OLE service will launch a property sheet in response to `QuickEdit` which looks and works similarly to all of those which get launched from the Enterprise Manager Console Navigator. This dialog should be mode-less but should dismiss after use. There may be multiple `QuickEdit` property sheets invoked for objects of your type at once, but there should be at most one for each object of your type.

6

Discovery Cache Integration

This chapter covers the discovery cache interface. It describes:

- Retrieving Nodes and Services
- Retrieving User-Defined Groups
- Service Types
- Discovering Services
- Discovery Cache API Reference

Retrieving Nodes and Services

The discovery cache interface allows your application to retrieve:

- The nodes and services available to the Console.
- The state of nodes and services.

An application can retrieve information about the databases, listeners, nodes, and third-party (external) service types that have been discovered by Oracle Enterprise Manager. You can retrieve both the names of these nodes and services and state information.

State information is only maintained in the discovery cache for nodes and services which have Up/Down events registered on them. If an Up/Down event is not registered for a node or service, the state is `VOXEXT_SERVICE_UNMONITORED`. A consequence of this is that no externally defined service types can be monitored in this release. If the Up/Down event is registered for a node or service, the status can be either `VOXEXT_SERVICE_UP` or `VOXEXT_SERVICE_DOWN`.

For an example of how to retrieve a list of nodes or services objects of a certain type, see `GetObjectList` of `CGetObjListDlg::DoGetObjectList` in the `dcobjlst.cpp` file. For an example of how to retrieve state of a node or service object, see `GetObjectState` of `CGetObjStateDlg::DoGetObjectState` in the `dcobjsta.cpp` file.

Retrieving User-Defined Groups

You can create groups in Console that contains nodes or services objects of the same type. An application can retrieve:

- Groups of a specified type
- Objects in a specified group, including subgroups
- Objects in a specified group, excluding subgroups, and their respective states
- Node name where the service is located.

For an example of how to retrieve a list of groups of a certain type, see `GetGroupsOfType` of `CGetGroupsOfTypeDlg::DoGetGroupsOfType` in the `dcgrptyp.cpp` file.

For an example of how to retrieve a list of objects, including subgroups, in a specified group, see `GetObjectsInGroup` of `CGetObjInGroupDlg::DoGetObjectsInGroup` in the `dcobjgrp.cpp` file.

For an example of how to retrieve a list of objects in a group with their states, see `GetUniqueServices` of `CGetUniqueSrvDlg::DoGetUniqueServices` in the `dcunqsrv.cpp` file. The list is flattened and objects only appear once in the list.

For an example of how to retrieve the node name where the service is located, see `GetServiceNode` of `CGetServiceNodeDlg::DoGetServiceNode` in the `dcsrvnod.cpp` file.

Service Types

Most of the discovery cache APIs, as well as many other APIs of other categories, use a VTS_BSTR to indicate service types. Service types can be those which are defined by Enterprise Manager itself, sometimes referred to as internally-defined service types, or those which third parties have defined themselves, sometimes referred to as externally-defined or user defined service types. The internally-defined service types used in this release are:

Table 6–1 Service Types

Type Value (from `voxtype.h`)	Service
VOXEXT_TYPE_AGENT	Oracle Intelligent Agent
VOXTYPE_TYPE_DATABASE	Oracle Database
VOXTYPE_TYPE_LISTENER	Oracle SQL*Net Listener
VOXTYPE_TYPE_NAMESERVER	Oracle Names Server
VOXTYPE_TYPE_NODE	Host machine
VOXTYPE_TYPE_OPS	Oracle Parallel Server
VOXEXT_TYPE_RDBDATABASE	Oracle Rdb Database
VOXEXT_TYPE_TRACE	Oracle Trace

The internal type names specified in the NT registry serve as the type values for externally-defined service types. There is no difference between the way the discovery cache treats internally and externally-defined service types or objects.

In almost all cases where you need to specify a service type as a parameter, you can use either internally or externally-defined types. For more information on externally-defined services, see Chapter 5, "Navigator and Map Integration".

Discovering Services

Every time the agent starts, it executes the `nmiconf.tcl` script which reads configuration files (`oratab`, `listener.ora`, and `tnsnames.ora`) and writes the `services.ora` file to the `$ORACLE_HOME\network\admin` directory. This text file contains information about services on the node. This information is used to populate the Navigator tree when retrieved by the Navigator Discovery option.

The `nmiconf.tcl` script can execute additional Tcl scripts written specifically to discover other services, such as the Oracle Web Server, on the node. If other scripts are used, they should be installed with `nmiconf.tcl` in the `$ORACLE_HOME\network\agent\config` directory (Windows platforms), and their names should be listed, one script per line, in the `nmiconf.lst` file located in the same directory. If errors occur during discovery, these are written to the `$ORACLE_HOME\network\log\nmiconf.log` file.

The Tcl scripts must be generate lines in the `services.ora` file of the form:

```
NewService = (servicetype, host, data)
```

This entry allows this service to be discovered by the Navigator discovery option. For example, to generate the following entry in the services.ora file:

```
MyNewService = (MY_SERVICE, MyHost, My new service)
```

you would create a Tcl script in following format:

```
set Parameters(MY_SERVICE) {ServiceType HostName Data};

set MyNewService "MyNewService";
set ServiceType($MyNewService) MY_SERVICE;
set HostName($MyNewService) "MyHost";
set Data($MyNewService) "'My new service'";

lappend ServiceNames $MyNewService;
```

After the Navigator Discovery wizard has discovered the `MyHost` node, a new folder named `MY_SERVICE` is added to the Navigator tree. `MyNewService` is located in the `MY_SERVICE` folder. See *GetObjectData* on page 6-7 for information on retrieving this information.

Discovery Cache API Reference

This section describes the external interfaces for the discovery cache system.

- GetGroupsOfType
- GetObjectData
- GetObjectList
- GetObjectsInGroup
- GetObjectState
- GetServiceNode
- GetUniqueServices

Common Parameters

These parameters are used with multiple discovery cache external interfaces and the descriptions are provided in this section.

Type
The service type of objects. See *Service Types* on page 6-3.

GroupName
The name of the user-defined group from which the services are to be extracted.

pData
Pointer to VARIANT containing retrieved data. Contains a SAFEARRAY. For example:

```
{
Name1, State1}
Name2, State2}
...
{Namex, Statex}
}
```

where *Namex* contains the node, service, or group name and *Statex* is an integer indicating:

```
VOXEXT_SERVICE_UP
VOXEXT_SERVICE_DOWN
VOXEXT_SERVICE_UNMONITORED
```

Note: These states are listed in the `voxext.h` file.

GetGroupsOfType

Purpose `GetGroupsOfType` retrieves a list of all the user-defined groups of a specified type.

Syntax `VT_BOOL GetGroupsOfType(VTS_BSTR Type, VTS_PVARIANT pData)`

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
Type	VTS_BSTR	IN
pData	VTS_PVARIANT	OUT

pData
pData is an array of the form:

```
{
{GroupName1},
{GroupName2},
...
{GroupNamex}
}
```

where *GroupNamex* is the name of the user-defined group.

Comments This function returns TRUE if it succeeds, otherwise it returns FALSE.

GetObjectData

Purpose `GetObjectData` retrieves data about objects in the Navigator tree.

Syntax
```
VT_BOOL GetObjectData(VTS_BSTR ServiceName,
                      VTS_BSTR ServiceType,
                      VTS_BSTR Location,
                      VTS_PSTR pServiceData);
```

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
ServiceName	VTS_BSTR	IN
ServiceType	VTS_BSTR	IN
Location	VTS_BSTR	IN
pServiceData	VTS_PBSTR	OUT

ServiceName
The name of the third-party discovered service for which you want to get the associated data.

ServiceType
The third-party service type that was used during the agent auto-discovery.

Location
The name of the node on which the service resides.

pServiceData
The data associated with the service. This is an arbitrary string that is a maximum of 1024 bytes.

Comments This function will only work for services that have been discovered from an agent using the agent's third-party discovery integration mechanism. See *Discovering Services* on page 6-4.

GetObjectList

Purpose `GetObjectList` retrieves a list of objects of a specified type.

Syntax
```
VT_BOOL GetObjectList(VTS_BSTR Type,
                      VTS_BSTR Location,
                       VTS_BSTR LocationType
                       VTS_BOOL WithAgent,
                       VTS_PVARIANT pData)
```

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
Type	VTS_BSTR	IN
Location	VTS_BSTR	IN
LocationType	VTS_BSTR	IN
WithAgent	VTS_BOOL	IN
pData	VTS_PVARIANT	OUT

Type
Table 6–2, "Type Parameter Behavior" describes the behavior of `GetObjectList` for different values of the parameter `Type`.

Table 6–2 Type Parameter Behavior

Value of Type	Behavior
LISTENER	Retrieves a list of all listeners on the node specified by Location. If Location is NULL returns a list of all listeners.
NAMESERVER	Retrieves a list of all name servers on the node specified by Location. If Location is NULL returns a list of all name servers.
DATABASE	Retrieves a list of all Oracle databases on the node specified by Location. If Location is NULL, returns a list of all Oracle databases.
NODE	Retrieves a list of all known nodes. Location is ignored.
Externally-defined types	Retrieves a list of externally-defined types. Location is ignored.

Location
Name of node or service object that the service is associated with.

LocationType
Type of service specified in Location field. If LocationType is `VOXTYPE_TYPE_NULL` and Location is an empty string, `GetObjectList` returns a list of all of the objects of type `ServiceType` in the discovery cache. Otherwise, the following situations are defined and no others.

Table 6–3 Location Types

ServiceType	LocationType	Meaning
VOXTYPE_TYPE_DATABASE	VOXTYPE_TYPE_LISTENER	Returns a list of databases associated with listener with name Location
VOXTYPE_TYPE_DATABASE	VOXTYPE_TYPE_OPS	Returns a list of database instances associated with the Parallel Server with name Location
Any (including external types)	VOXTYPE_TYPE_NODE	Returns a list of services of type ServiceType associated with node with name Location

WithAgent
If TRUE, `GetObjectList` returns a list of services of the specified type that reside on nodes with agents

pData
pData is an array of the form:

```
{
{Name1, State1},
{Name2, State2},
...
{Namex, Statex}
}
```

where *Namex* contains the node, service, or group name and *Statex* is an integer indicating:

```
VOXEXT_SERVICE_UP
VOXEXT_SERVICE_DOWN
VOXEXT_SERVICE_UNMONITORED
```

Comments This function returns TRUE if it succeeds, otherwise it returns FALSE.

GetObjectsInGroup

Purpose `GetObjectsInGroup` retrieves a list of the objects in the specified group.

Syntax
```
VT_BOOL GetObjectsInGroup(VTS_BSTR GroupName,
                  VTS_PVARIANT pData)
```

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
GroupName	VTS_BSTR	IN
pData	VTS_PVARIANT	OUT

pData
pData is an array of the form:

```
{
{Name1, Group_Flag1},
{Name2, Group_Flag2},
...
{Namex, Group_Flagx}
}
```

where *Namex* contains the node, service, or group name and *Group_Flagx* is set to TRUE if *Namex* is a group name, otherwise FALSE.

Comments A user-defined group may contain other groups. The list returned by `GetObjectsInGroup` in pData may contain other group names, which are designated by `GROUP_FLAG` set to TRUE.

This function returns a TRUE if it succeeds, otherwise it returns FALSE.

GetObjectState

Purpose `GetObjectState` retrieves the state of a specified node or service.

Syntax
```
VT_BOOL GetObjectState(VTS_BSTR Type,
                       VTS_BSTR Name,
                       VTS_BOOL Group,
                       VTS_PI2 pReturnState)
```

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
Type	VTS_BSTR	IN
Name	VTS_BSTR	IN
Group	VTS_BOOL	IN
pReturnState	VTS_PI2	OUT

Group
Determines whether the object is a group or not.

pReturnState
The state of the node, service, or group

```
VOXEXT_SERVICE_UP
VOXEXT_SERVICE_DOWN
VOXEXT_SERVICE_UNMONITORED
```

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE.

GetServiceNode

Purpose GetServiceNode retrieves the node name where the service is located.

Syntax
```
VTS_BOOL GetServiceNode(VTS_BSTR ServiceName,
                        VTS_BSTR ServiceType,
                        VTS_PBSTR pNodeName);
```

Parameters

Name	Type	Mode
ServiceName	VTS_BSTR	IN
ServiceType	VTS_BSTR	IN
pNodeName	VTS_BSTR	OUT

ServiceName
Name of service you want to find the node for.

ServiceType
Type of objects to be retrieved. These types are listed in the `voxtype.h` file. See Table 6–1, "Service Types".

pNodeName
Retrieves name of node where the service identified by `ServiceName` is located.

Comments An example is in the `dcsrvnod.cpp` file.

GetUniqueServices

Purpose `GetUniqueServices` retrieves all services within a group, including those within subgroups, without duplication of services.

Syntax
```
VT_BOOL GetUniqueServices(VTS_BSTR GroupName,
                VTS_PVARIANT pData)
```

Parameters See *Common Parameters* on page 6-5.

Name	Type	Mode
GroupName	VTS_BSTR	IN
pData	VTS_PVARIANT	OUT

pData
pData is an array of the form:

```
{
{Name1, State1}
{Name2, State2}
...
{Namex, Statex}
}
```

where *Namex* contains the node, service, or group name and *Statex* is an integer indicating:

```
VOXEXT_SERVICE_UP
VOXEXT_SERVICE_DOWN
VOXEXT_SERVICE_UNMONITORED
```

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE

7

Job Scheduling Integration

This chapter covers the Job Scheduling system Interface. It describes:

- Submitting a Job
- Deleting a Batch Job
- Job Notification
- Job Scripting
- Job Scheduling API Reference
- VoxJobNotifyUnpacker Class and Methods

Submitting a Job

The Job Scheduling system interface allows you to submit both batch and interactive jobs. The steps for creating and submitting a job depend on whether the job is a batch job or an interactive job. See *Submitting a Batch Job* on page 7-2 or *Submitting an Interactive Job* on page 7-3.

Submitting a Batch Job

A batch job is one that is submitted to the Job Scheduling system. The Enterprise Manager Console gathers the job's details and schedules the job with the agent on the node on which the job will run. A batch job can be scheduled to run more than once according to a specified schedule. All batch jobs can be observed from the Console.

Note: Unless it is *read-only*, an externally-submitted batch job can be deleted from the Console.

To submit a batch job:

1. Create a OraTcl job script for the job. See Chapter 9, "Jobs and Events Scripts" for more information about writing job scripts.
2. Copy the job's script to an accessible directory. Place a copy of the script in a directory on a file system accessible to the Console.
3. Create and initialize a job object, as described in *Initialize* on page 7-9. When submitting a new batch job, set `JobID` to zero and `Batch` to TRUE in the call to `Initialize`. If you want to create a read-only job, set the `ReadOnly` parameter to TRUE.
4. Submit the job details, setting the destinations first. The functions available for specifying job details are:
   a. `SetDestinationsEx` to submit the services or nodes on which to run the job, as well as to specify parameters for each destination and the names of the input files. See *SetDestinationsEx* on page 7-12.
   b. `SetScript` to submit the name of the script for the job. See *SetScript* on page 7-19.
   c. `SetJobName` to provide a clear-text name for the Console to display when referring to the job. See *SetJobName* on page 7-13.
   d. `SetSchedule` to set the job schedule. See *SetSchedule* on page 7-15.

e. `SetNotificationObjectProgID` to submit the class name of the OLE automation server to receive notifications about the job (optional). See *SetNotificationObjectProgID* on page 7-14.

f. `SetCredentials` to specify the host username and password for the job. See *SetCredentials* on page 7-11.

5. Commit the job and store the `JobID` if you plan to coordinate job notifications. See *Commit* on page 7-8.

6. Process the job notifications as they return if `SetNotificationObjectProgID` was called.

Submitting an Interactive Job

An interactive job is one that is submitted and executed immediately. Typically these are jobs that require the application user to wait until the job is completed. When an interactive job is submitted, the communication daemon tries to contact the agent immediately. If the daemon cannot contact the agent, the job fails.

Interactive jobs are not logged in the same way as batch jobs. From the Console user's point of view these jobs do not exist. These jobs provide a service for other applications, such as database backup or recovery. Only the third-party application user sees that the application has performed an action on a remote node.

To submit an interactive job:

1. Create a OraTcl job script for the job. See Chapter 9, "Jobs and Events Scripts" for more information about writing job scripts.

2. Copy the job's script to an accessible directory. Place a copy of the script in a directory on a file system accessible to the Console.

3. Create and initialize a job object. Initialize a job object, as described in *Initialize* on page 7-9. Because you are submitting an interactive job, set `JobID` to zero, and `Batch` and `ReadOnly` to FALSE in the call to `Initialize`.

4. Submit the job details, setting destinations first. The functions available for specifying job details are:

   a. `SetDestinationsEx` to submit the service or node on which to run the job, as well as to specify parameters for each destination and the names of the input files. See *SetDestinationsEx* on page 7-12.

   b. `SetScript` to submit the fully qualified name of the script. See *SetScript* on page 7-19.

c. `SetNotificationObjectProgID` to submit the name of the OLE automation server to receive notifications about the job. See *SetNotificationObjectProgID* on page 7-14.

Note: `SetNotificationObjectProgID` is required for interactive jobs.

d. `SetCredentials` (optional) to submit the username and password to be used for the job. See *SetCredentials* on page 7-11.

Note: `SetCredentials` is optional for interactive jobs. If you do not explicitly set the job credentials, the Console submits the job using the preferred credentials specified for the target destination.

5. Commit the job. For an interactive job, the call to the function `Commit` causes the daemon to contact the appropriate agent and submit the interactive job immediately. If the daemon successfully passes the job to the agent, it passes back a `VOXJOB_STATUS_SCHEDULED` notification. If the daemon is unsuccessful, it passes back an error code. See *Commit* on page 7-8.

Deleting a Batch Job

Only batch jobs can be deleted. If a job is currently running when it is deleted, the agent will kill the process running the job. If you are deleting a job that was created with a service to be notified when the job's status changed, then you will receive a notification with the status `VOXJOBNT_STATUS_DELETED` when the job has been deleted. To delete a batch job:

1. Initialize a job object. Initialize a job object, as described in *Submitting a Batch Job* on page 7-2. Set `JobID` to the ID of the job you want to delete and `Batch` to TRUE in the call to `Initialize`.

2. Delete the job. Call the function `DeleteJob` to delete a job.

Note: Interactive jobs cannot be deleted.

Job Notification

In order to submit an interactive job or to receive messages about status changes for batch jobs, an OLE automation server that exposes a `JobNotification` method must be exposed by your application.

If you configure the job to receive notifications, then the job's output and any output parameters can be returned to you. If you do not, the Job Scheduling system will not notify you programmatically of any changes in the job's status, nor will it return any output parameters to you.

Extracting Job Notification Information

Use the helper function `VoxJobNotifyUnpacker` to extract all the information from the VARIANT passed to your application's `JobNotification` method.

Flushing the Job Queue

When an application is launched, use the function `RegisterApplication` to inform the Job Scheduling system to send any notifications that have been pending since an application had last run.

Who Is Notified

Just as in event notifications (see *Notification* on page 8-3), jobs are referenced by `ProgID` and current Console user. This ensures that a user receives only notifications from the jobs that the user submits and that each application maintains its own notification queue. The application does not have to pass the console username because the Console already knows it.

Job Notification Messages

When you submit a job, you will receive one of the following notifications:

- `VOXEXT_JOB_STATUS_SUBMITTED` if the submission succeeds.
- `VOXEXT_JOB_STATUS_FAILED` if it did not.

When a job initiates, you will receive a `VOXEXT_JOB_STATUS_STARTED` notification. Next you will receive one of the following notifications:

- `VOXEXT_JOB_STATUS_COMPLETED` if the job succeeded.
- `VOXEXT_JOB_STATUS_FAILED` if it failed.

This sequence will continue to repeat itself for batch jobs which have repeating schedules. After the last scheduled job execution completes, whether successfully or not, you will receive a VOXEXT_JOB_STATUS_EXPIRED notification, which is the last notification in the sequence for any given job. Note that you still get VOXEXT_JOB_STATUS_EXPIRED notifications for one-time or immediate jobs, including all interactive jobs.

After the successful deletion of a batch job, you receive a VOXEXT_JOB_STATUS_CANCELED notification.

Intermediate job status is not a part of the typical sequence of job notifications. It is provided for applications which want to record the progress or intermediate status of a job in the middle of its execution. The VOXEXT_JOB_STATUS_INTERMED notification is sent only when a job script containing the orajobstat OraTcl verb is invoked. For information on orajobstat, in see *OraTcl Functions and Parameters* on page 9-12. For an example of the use of orajobstat, see the intermed.tcl sample script in the ORACLE_HOME\SYSMAN\SDK\TCL directory.

Job Scripting

Every job is a Tcl script that is executed by the Oracle Intelligent Agent on a particular node. Jobs cannot be executed on nodes which do not have an agent running. Jobs are categorized in the Job Scheduling system of the Console according to different service types, such as database or listener jobs. However, all jobs are fundamentally node jobs and categories simply signify the script's effects. The credentials specified using the SetCredentials API are the node credentials with which the Tcl script executes.

The Tcl script does not get executed as if it were executed directly in the Tcl shell on that node. The Tcl scripts are framed within a Tcl program called the Master Tcl Script by the Job Scheduling system before the script gets sent to the agent. The Master Tcl script sets up the majority of the environment needed for jobs sent from the Enterprise Manager Console's user interface. However, there are some side effects of the Master Tcl Script environment that you can take advantage of. For example, the following variables are defined for your use:

Table 7–1 Tcl Variables

Variable	Meaning
$SMP_USER	user name of job credential
$SMP_PASSWORD	password of job credential
$SMP_SERVICE	service name of job destination

If a job needs to send files other than the actual Tcl script, send them as input files. You can specify these files using the `SetScript` API. One example is a Tcl job script that executes SQL and uses input files that contain SQL scripts. Another example is a job that invokes SQL*Plus with a SQL script, or Export with a specification file.

Input files get copied to the destination's node from the location specified on the console file system. You can find out the names of the input files on the destination's file system by referencing the OraTcl array `oramsg`(orainput). `oramsg`(orainput), defined for jobs only, is a Tcl list that contains the names of the job's input files. For more information on `oramsg`, see *oramsg Elements* on page 9-6.

Sending Jobs that Execute SQL*Plus Scripts

The file `exec_sql.tcl` is included in the SDK to provide jobs that execute SQL*Plus scripts. To send a job which does this, specify `exec_sql.tcl` with a fully qualified path as the script for a specific job in the `ScriptName` parameter to the `SetScript` API, and pass in the fully qualified path name of the SQL*Plus script to be executed as the sole input file in the `InputFilenames` parameter. The output of the SQL*Plus script will be sent to the application via the `JobNotification` mechanism as the output string in the `VOXEXT_JOB_STATUS_COMPLETED` notification.

Job Scheduling API Reference

The following section describes the external interface calls for the Job Scheduling system.

- Commit
- DeleteJob
- Initialize
- JobNotification
- RegisterApplication
- SetCredentials
- SetDestinationsEx
- SetNotificationObjectProgID
- SetSchedule
- SetJobName
- SetScript

Commit

Purpose `Commit` causes the job object to schedule the job.

Syntax `VT_BOOL Commit(VTS_PI4 pJobID)`

Parameters

Name	Type	Mode
`pJobID`	`VTS_PI4`	`IN/OUT`

pJobID
The unique identifier for a job.

Comments If `Commit` is not called, the job is not submitted. This function returns a TRUE if it succeeds, otherwise it returns FALSE. `pJobID` is passed out after the commit has succeeded.

For an example illustrating how to use `Commit`, see `CJobBatchCreateDlg::DoCommit` in the `jobbatch.cpp` file.

DeleteJob

Purpose `DeleteJob` allows for the deletion and the removal of the job from the remote nodes schedule.

Syntax `VT_BOOL DeleteJob()`

Parameters None

Comments Interactive jobs cannot be deleted. This function returns a TRUE if it succeeds, otherwise it returns FALSE.

Note: `Initialize` must be called before `DeleteJob`.

Initialize

Purpose `Initialize` initializes a newly-created a job object.

Syntax
```
VT_BOOL Initialize(VTS_I4 JobID,
                   VTS_BOOL Batch,
                   VTS_BOOL ReadOnly)
```

Parameters

Name	Type	Mode
JobID	VTS_I4	IN
Batch	VTS_BOOL	IN
ReadOnly	VTS_BOOL	IN

JobID
The unique identifier for a job. Set to zero to create a new job.

Batch
Indicates whether the job is an interactive or batch job. This parameter only applies if `JobID` is zero.

ReadOnly
Indicates whether the job can be modified by the Console user or not. This parameter only applies if the job is batch and `JobID` is zero.

Comments If you are initializing the job object in order to create a job, set `JobID` to zero. If you are initializing the job object in order to delete a job, then set `JobID` to the ID of the job you plan to delete. This is the `JobID` returned as an out parameter from the `Commit` call when the job was submitted.

You cannot delete an interactive job. Therefore, the parameter `Batch` applies only when `JobID` is zero indicating that the job object should be placed in create mode.

Set the `ReadOnly` parameter to TRUE if you do not want a Console user to be able to delete the job. ReadOnly jobs cannot be altered by the Console, but they can be deleted programmatically through API calls.

This function returns a TRUE if it succeeds, otherwise it returns FALSE.

For an example illustrating the steps for initializing a job object, see `CSmpSrvDlg::GetJobObject` in the `smpsrvdl.cpp` file.

JobNotification

Purpose The Job Scheduling system uses this method to notify your application about job status changes. This interface gets called by the daemon directly.

Syntax `VT_VOID JobNotification(VTS_VARIANT Notification)`

Parameters

Name	Type	Mode
`Notification`	`VTS_VARIANT`	`IN`

Notification
This contains all the notification information that is returned by the daemon.

Comments In order to receive messages about job status changes, an application must contain an OLE automation server that exposes a `JobNotification` method. When the status of a job changes, the Intelligent Agent notifies the Communication Daemon, which calls this interface of the application. If the job is a batch job, the change in job status is also reflected in the Console.

The VOX library provides a class, `VoxJobNotifyUnpacker` to make the unpacking of the VARIANT passed to the `JobNotification` interface easy and error-free. Simply declare an object of this class passing the VARIANT to the constructor, and then call any of the following class member functions to access the data. For related information, see `VoxJobNotifyUnpacker`.

RegisterApplication

Purpose `RegisterApplication` is used to flush queued job and event notifications.

Syntax `VT_BOOL RegisterApplication(VTS_BSTR ProgID):`

Parameters

Name	Type	Mode
ProgID	VTS_BSTR	IN

ProgID
The name of the OLE service which implements the `JobNotification` API.

Comments Whenever the OLE service that exposes the `JobNotification` API is temporarily unavailable, such as when the application is not running, the communication daemon queues its job notifications. In order to retrieve the queued notifications, call the `RegisterApplication` API. Typically, you will call this API when the application or OLE service initializes.

Note: You must call `RegisterApplication` to resend job notifications that have been queued.

SetCredentials

Purpose `SetCredentials` is used to set operating systems credentials, username and password, for the job.

Syntax `VT_BOOL SetCredentials(VTS_BSTR Username, VTS_BSTR Password)`

Parameters

Name	Type	Mode
Username	VTS_BSTR	IN
Password	VTS_BSTR	IN

Username
The username for the job.

Password
The password for the job.

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE.

Note: `SetCredentials` is optional. If you do not explicitly set the job credentials, the Console submits the job using the preferred credentials specified for the target destination.

For an example illustrating how to use `SetCredentials`, see `CJobCredsDlg::DoSetCredentials` in the `jobcreds.cpp` file.

SetDestinationsEx

Purpose `SetDestinationsEx` submits the services or nodes against which the job is to be run and the destination-specific parameters.

Syntax
```
VT_BOOL SetDestinationsEx(VTS_BSTR DestinationType,
                          VTS_VARIANT Destinations)
```

Parameters

Name	Type	Mode
`DestinationType`	`VTS_BSTR`	`IN`
`Destinations`	`VTS_VARIANT`	`IN`

DestinationType
For information on the types, see *Service Types* on page 6-3.

Destinations
A VTS_VARIANT containing a two-dimensional SAFEARRAY of VT_BSTR of the form:

```
{
{Name1, ParameterList1},
{Name2, ParameterList2},
...
{Name n, ParameterList n}
}
```

where *Namex* is a VT_BSTR containing the service or node name and *ParameterListx* is a VT_BSTR which contains a curly-brace delimited list of arguments. For example, `Destinations` could be set to:

```
system1, {scott/tiger@ora8db.world}
```

To pass a NULL parameter just use two curly braces that delimit nothing. There can be only one parameter list per destination.

Comments The destination types are defined in the `voxtype.h` file. There must be at least one destination. This function returns a TRUE if it succeeds, otherwise it returns FALSE.

For an example illustrating how to use `SetDestinationsEx`, see `CJobDestDlg::DoSetDestinationsEx` in the `jobcreds.cpp` file.

SetJobName

Purpose `SetJobName` associates a clear-text name with the job. This text is used by the Console when it displays the job's name.

Syntax `VT_BOOL SetJobName(VTS_BSTR JobName)`

Parameters

Name	Type	Mode
JobName	VTS_BSTR	IN

JobName
The name by which the Console refers to the job object.

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE.

Note: This function should only be used for batch jobs.

For an example illustrating how to use `SetJobName`, see `CJobNameDlg::DoSetJobName` in the `jobnamed.cpp` file.

SetNotificationObjectProgID

Purpose `SetNotificationObjectProgID` is used to submit the name of your application's OLE automation server, so that the Job Scheduling system can send notification information to your application when a job's status changes.

Syntax `VT_BOOL SetNotificationObjectProgID(VTS_BSTR ProgID)`

Parameters

Name	Type	Mode
`ProgID`	`VTS_BSTR`	`IN`

ProgID
The name of the OLE automation server that will be informed when any status changes occur for the job, such as `Smpsrv.Document`. The server must be declared to be the active object in order for the Communication Daemon to locate the `JobNotification` interface.

Comments See *Job Notification* on page 7-5 for a description of the messages returned when the job state has changed. This function returns a TRUE if it succeeds, otherwise it returns FALSE.

Note: This call is optional for a batch job. For an example illustrating how to use `SetNotificationObjectProgID`, see `CJobServiceDlg::DoSetNotificationObjectProgID` in the `jobservi.cpp` file.

For related information, see `JobNotification`.

SetSchedule

Purpose `SetSchedule` sets the date, time, and frequency at which to run the job.

Syntax `VT_BOOL SetSchedule(VTS_BSTR Schedule)`

Parameters

Name	Type	Mode
`Schedule`	`VTS_BSTR`	`IN`

Schedule
The `schedule` string is divided into several clauses:

- Repeat Frequency
- Start Time
- End Time
- Time Zone

Repeat Frequency Clause
This clause is mandatory and specifies the frequency with which the job is executed. There are six different repeat modes, specified by the /R keyword. Some modes require a frequency to be specified as well. This is accomplished with the /F or /ON keywords.

All repetitions, except immediate execution, execute initially at the start time and date. If an end time and date are specified, the job ends on the last repetition which occurs before or on that time and date. All times are interpreted relative to the time zone specified.

For those repeat modes which only specify what days an execution is scheduled for, that execution will take place at the time specified by start time of that day. End times need not be specified for such schedules. Multiple repeat clauses can be specified. If this is done, jobs will be executed whenever any of the repeat clauses specify.

.Repeat Mode: Immediate

Table 7–2 Repeat Mode: Immediate

Format	Interpretation	Examples
/R=I	Execute immediately.	/R=I

Repeat Mode: Once

Table 7–3 Repeat Mode: Once

Format	Interpretation	Examples
/R=O	Execute once at start date and time.	/R=O

Repeat Mode: Every Time Interval

Table 7–4 Repeat Mode: Every Time Interval

Format	Interpretation	Examples
/R=H /F=HH:MM:SS	Execute repeatedly every specified interval. All fields must be specified.	/R=H /F=02:30:00 Execute every 2 hours, 30 minutes, 0 seconds.

Repeat Mode: Every Day Interval

Table 7–5 Repeat Mode: Every Day Interval

Format	Interpretation	Examples
/R=D /F=<# of days>	Execute once every specified number of days.	/R=D /F=2 Execute once every two days.

Repeat Mode: Every Day of Week Interval

Table 7–6 Repeat Mode: Every Day of Week Interval

Format	Interpretation	Examples
/R=W /ON=Sun, Mon, ..., Sat where Day is taken from {Sun, Mon, Tue, Wed, Thu, Fri, Sat}	Execute every specified day of the week.	/R=W /ON=Mon, Wed, Fri Execute every Monday, Wednesday, and Friday.

Repeat Mode: Every Date of Month

Table 7–7 Repeat Mode: Every Date of Month

Format	Interpretation	Examples
/ R=M /ON=1, 2, ..., 31 where Date is a number from 1 to 31 representing a date of the month.	Execute on every specified date of the month.	/R=M /ON=12 Execute every 12th of the month. /R=M /ON=7, 15, 30 Execute every 7th, 15th, and 30th of the month.

Start Time Clause

This clause specifies the start date and start time for the schedule. Inclusion of a start time clause is mandatory for all repeat modes except immediate execution. Time is interpreted relative to the time zone specified.

For those repeat modes which only specify what days an execution is scheduled for, that execution will take place at the time specified by start time of that day.

Table 7–8 Start Time Clause

Format	Interpretation	Examples
/SD=MM/DD/YY /ST=HH:MM For years on or after 2000, use YYYY.	Start date of specified month, date, and year. Start time of specified time on 24 hour clock.	/SD=04/17/98 /ST=13:30 Execute starting on April 17, 1998 at 1:30 PM

End Time Clause

This clause specifies the end date and end time for the schedule. Time is interpreted relative to the specified time zone. Use of an end time clause is mandatory for all modes except immediate and one-time execution. Use an end date in the distant future for an indefinite execution. The end time phrase of this clause can be omitted for all but the Every Time Interval Repeat Mode. For those repeat modes which only specify what days an execution is scheduled for, that execution ceases at the time specified by start time of that day if no end time is specified. Otherwise, the execution will cease on the execution scheduled just before the end time and date.

Table 7–9 End Time Clause

Format	Interpretation	Examples
/ED=MM/DD/YY /ET=HH:MM For years on or after 2000, use YYYY.	End date of specified month, date, and year. End time of specified time on 24 hour clock.	/ED=04/20/2000 /ET=09:00 Execute ending on April 20, 2000 at 9:00 AM. /ED=12/01/2001 Execute ending on December 1, 2001 at start time.

Time Zone Clause

This clause is optional and specifies the time zone for which all times should be interpreted. Omission of this clause means that the string should be interpreted relative to the time zone of the host on which the computation is being performed.

Table 7–10 Time Zone Clause

Format	Interpretation	Examples
/GMT+offset	Times should be interpreted relative to Greenwich Mean Time, offset as specified. Offsets can be positive or negative, and can also be expressed as decimals.	/GMT Times are Greenwich Mean Time. /GMT-2 Times are two hours behind Greenwich Mean Time (GMT). /GMT+5.5 Times are 5.5 hours ahead of GMT.

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE. This function should only be used for batch jobs. Interactive jobs execute immediately. For an example illustrating how to use `SetSchedule`, see `CJobSchedDlg::DoSetSchedule` in the `jobsched.cpp` file.

SetScript

Purpose `SetScript` submits the Tcl script which defines the job, plus a list of the input files for the job.

Syntax
```
VT_BOOL SetScript(VTS_BSTR ScriptName,
                  VTS_VARIANT InputFilenames)
```

Parameters

Name	Type	Mode
`ScriptName`	`VTS_BSTR`	`IN`
`InputFilenames`	`VTS_VARIANT`	`IN`

ScriptName
A fully-qualified path name of a Tcl script such as, `c:\orant\sysman\sdk\tcl\exec_sql.tcl`

InputFilenames
A VTS_VARIANT containing a SAFEARRAY of the form:

```
{
{Filename1}
{Filename2}
...
{Filenamen}
}
```

where *Filenamex* is a VT_BSTR containing the fully qualified filenames of the input files, such as `c:\orant\sysman\sdk\samples\smpsrv\sample.tcl`.

Comments This function returns a TRUE if it succeeds, otherwise it returns FALSE. `oramsg(orainput)`, defined for jobs only, is a Tcl list that contains the names of the job's input files. For more information on `oramsg`, see *oramsg Elements* on page 9-6.

For an example illustrating how to use SetScript, see `CJobScriptDlg::DoSetScript` in the `jobscrip.cpp` file.

VoxJobNotifyUnpacker Class and Methods

The class `VoxJobNotifyUnpacker` extracts job notification information from the VARIANT `vtNotification` parameter of the application's `JobNotification` interface. Call any of the following class member functions to access the data.

- GetDate
- GetError
- GetJobID
- GetNode
- GetOutput
- GetStatus

For an example illustrating how to use `VoxJobNotifyUnpacker`, see the `smpsrdoc.cpp` file. For related information, see *JobNotification* on page 7-10.

GetDate

Purpose `GetDate` returns textual representation of time and date of notification.

Syntax `const CString& GetDate() const;`

Comments None.

GetError

Purpose `GetError` returns an error code in case of error condition.

Syntax `ULONG GetError() const;`

Comments `GetError` returns an error code as follows:

```
VOXJOB_ERROR_QUEUE - queue facility error
VOXJOB_ERROR_FILE - file operation error
VOXJOB_ERROR_MM memory - manager error
VOXJOB_ERROR_TCL - Tcl error
VOXJOB_ERROR_UNIQUE - job name is not unique
VOXJOB_ERROR_JNOTFOUND - job is not found
VOXJOB_ERROR_SNOTFOUND - Tcl script is not found
VOXJOB_ERROR_MANDATORY - mandatory input is missing
VOXJOB_ERROR_MAXLEN - exceeded maximum string length
VOXJOB_ERROR_SCHEDULE - scheduling error
VOXJOB_ERROR_MAXINP - exceeded maximum number of input files
VOXJOB_ERROR_NOUSER - no such user
VOXJOB_ERROR_INTERUPT - outstanding Tcl exists, cannot interrupt
VOXJOB_ERROR_CONNECTION - failed to connect to agent
```

GetJobID

Purpose `GetJobID` returns the ID of job that the notification corresponds to.

Syntax `ULONG GetJobID() const;`

Comments Returns ID of 0 if unpacking of VARIANT failed.

GetNode

Purpose `GetNode` returns name of node upon which job was scheduled for execution.

Syntax `const CString& GetNode() const;`

Comments None.

GetOutput

Purpose `GetOutput` returns output for notification.

Syntax `const CString& GetOutput() const;`

Comments Only used for notifications with status `VOXEXT_JOB_STATUS_COMPLETED`, `VOXEXT_JOB_STATUS_FAILED`, or `VOXEXT_JOB_STATUS_INTERMED`. In the latter two cases, the output is what would appear as stdout for the Tcl of your job. Intermediate job status output is under user control. See *Job Notification Messages* on page 7-5.

GetStatus

Purpose `GetStatus` returns the status of this notification.

Syntax `ULONG GetStatus() const;`

Comments `GetStatus` returns the status as follows:

```
VOXEXT_JOB_STATUS_NONE
VOXEXT_JOB_STATUS_SUBMITTED
VOXEXT_JOB_STATUS_STARTED
VOXEXT_JOB_STATUS_CANCELED
VOXEXT_JOB_STATUS_FAILED
VOXEXT_JOB_STATUS_COMPLETED
VOXEXT_JOB_STATUS_EXPIRED
VOXEXT_JOB_STATUS_INTERMED
```

Note: The statuses are listed in the `voxext.h` file.

8

Event Management Integration

This chapter describes integration with the Event Management system (EMS). Topics include:

- Levels of Integration
- Client-Side Integration
- Server-Side Integration
- Event Management System APIs
- VoxEventNotifyUnpacker

Levels of Integration

There are two levels of integration with the Event Management System. They are:

- Application support for event registration and notification.
- Agent interfaces for generation of events by third-party services.

These can be implemented separately or together, depending on your needs.

Client-Side Integration

An application must register interest in an event by calling the `RegisterEventInterest` API. You must register an event set with the Console Event Management system before calling `RegisterEventInterest`.

For the Console to monitor third-party events, you need to create an event set with the Event Management system. When you create the event set, check the "Accept Third-Party Events" box in the Event Set General page. You do not need to enter any information in the Events or Parameters pages. After the event set is created, register it on all destinations where an application registers interest. This registration will monitor all third-party events on the specified destination.

Uniqueness of Registration

All event registrations have to be unique, and EMS will return an error if an application tries to register the same event at the same node more than once. The uniqueness is defined by the application, event name, and the system name. However, the string "*" is an exception and can be used as a wildcard event name or destination.

An application may register "*", and "*/user/rdbms/fault/event1*" at node "*smpsun14*". EMS handles these as two registrations. If the application tries to register the "*" or the "*/user/rdbms/fault/event1*" event again at node "*smpsun14*", EMS raises an error. However, the application can register the same event at other systems. For information on the format of event names, see *eventname* on page 9-28.

For a particular event, an application will be notified only once. For example, in Table 8–1, "A Sample Event Registration Database", if "*/user/rdbms/fault/event1*" fires at "*smpsun14*", the application will be notified only once.

Table 8–1 A Sample Event Registration Database

Application	Event Name	System Name
DbApp	/user/rdbms/fault/event1	smpsun14
DbApp	/user/rdbms/fault/event2	smpsun14
DbApp	*	*
DbApp	*	smpsun14
DbApp	/user/rdbms/fault/event1	*
DbApp	*	smpsun15

Notification

In order for an application to be notified of an event you must expose the function `EventNotification`. This function is called whenever an event you have registered interest in is triggered. The `VoxEventNotifyUnpacker` class in the `vox.dll` is provided to unpack the parameters from the variant returned by `EventNotification`.

If an application is not active when the event is triggered, the Event Management System queues the event notifications. Also, the OLE object service must be declared the active object. When the application next comes up and calls `RegisterApplication` for the user who owns the event, all the queued events are forwarded.

Who Is Notified

Just as in job notifications (see *Who Is Notified* on page 7-5), events are referenced by `ProgID` and current Console user. This means that a user gets only notifications for the events that the user submits. Your server application does not need to pass the username because the Console already knows it.

Discovery Cache Event Management

The Event Management System sets up the discovery cache event-level based on the internally-defined event name that is returned by the intelligent agent. For example, if the event returned is `/user/rdbms/fault/event1`, then the Event Management System tries to locate the object name as a database, and updates the discovery cache, resulting in the map displaying the appropriate color.

Event Interest

There is one API to register interest in events: `RegisterEventInterest`. There are two API calls that allow you to cancel interest in events: `CancelEventInterest` and `CancelAllEvents`

Server-Side Integration

An *unsolicited* event is an event that is not discovered by the agent running a Tcl script itself, which is the normal way for events to get triggered. There are a number of reasons why this might occur, such as:

- A third-party, such as the operating system vendor or a systems management vendor, might discover the condition and want to have it appear on the Enterprise Manager Console. A key feature of Enterprise Manager is that it is open to third parties.
- An OEM job might discover the condition. An example of this is the Software Management component of Enterprise Manager, where a job is going to install a new package on the host, and then wants to report the event "new package discovered", rather than relying on the job reporting mechanisms. This provides uniform handling of the condition when there is a new package on this host.

You can raise unsolicited events with the Oracle Intelligent Agent using:

- The `orareportevent` OraTcl verb in Tcl scripts.
- The `oemevent` executable located in `oracle_home\bin` on a Unix platform or `oracle_home\agentbin` on a Windows NT machine.

The syntax for `orareportevent` and `oemevent` is:

```
orareportevent eventname object severity message [results]
oemevent eventname object severity message [results]
```

The parameters are the same for both except for `severity`. See *orareportevent* on page 9-28.

Event Management System APIs

The Event Management system has the following API calls:

- CancelAllEvents
- CancelEventInterest
- EventNotification
- RegisterApplication
- RegisterEventInterest

Common Parameters

These parameters are used in multiple EMS API calls and are described in this section.

ProgID
Identifies the OLE service that is interested in the event.

EventName
The name of the event that the application is interested in. This can be "*", which means all events.

Destination
The name of the system on which the event occurs. This can be "*", which means all destinations that have agents

CancelAllEvents

Purpose `CancelAllEvents` removes all event registration entries for the application specified.

Syntax `VT_BOOL CancelAllEvents(VTS_BSTR ProgID)`

Parameters See *Common Parameters* on page 8-5.

Name	Type	Mode
ProgId	VTS_BSTR	IN

Comments None.

CancelEventInterest

Purpose `CancelEventInterest` cancels interest for the specified event.

Syntax
```
CancelEventInterest(VTS_BSTR ProgID,
                    VTS_BSTR EventName,
                    VTS_BSTR Destination)
```

Parameters See *Common Parameters* on page 8-5.

Name	Type	Mode
ProgId	VTS_BSTR	IN
EventName	VTS_BSTR	IN
Destination	VTS_BSTR	IN

Comments This is a one to one match with the `RegisterEventInterest` function.

Examples For example, the following commands register interest in events:

```
RegisterEventInterest("Smpsrv.Document","/user/rdbms/fault/event1","system1");
RegisterEventInterest("Smpsrv.Document","*","*");
The following command removes the second entry alone.
CancelEventInterest("Smpsrv.Document","*","*")
```

To cancel the first event, the application has to call:

```
CancelEventInterest("Smpsrv.Document","/user/rdbms/fault/event1","system1")
```

EventNotification

Purpose The Communication Daemon calls this function to notify an application when a registered event has been triggered.

Syntax `VT_VOID EventNotification(VTS_VARIANT Notification)`

Parameters

Name	Type	Mode
Notification	VTS_VARIANT	IN

Notification
A VARIANT containing information regarding the event's name, node, object, date, and severity. Use the unpacker functions to access the information

Comments Oracle provides a variant unpacker class, `VoxEventNotifyUnpacker`, to ease the unpacking of the parameters from the variant. This is present in the `vox.dll`.

RegisterApplication

Purpose `RegisterApplication` is used to flush queued notifications.

Syntax `VT_BOOL RegisterApplication(VTS_BSTR ProgID)`

Parameters

Name	Type	Mode
ProgId	VTS_BSTR	IN

ProgID
The name of the OLE service which implements the `JobNotification` API.

Comments None.

RegisterEventInterest

Purpose `RegisterEventInterest` is used to register an application's interest in any events.

Syntax
```
VT_BOOL RegisterEventInterest(VTS_BSTR ProgID,
                VTS_BSTR EventName,
                VTS_BSTR Destination)
```

Parameters See *Common Parameters* on page 8-5.

Name	Type	Mode
ProgId	VTS_BSTR	IN
EventName	VTS_BSTR	IN
Destination	VTS_BSTR	IN

Comments If an application wants to register an interest in multiple events, it has to call this API multiple times with different event names. You can use "*" to register interest in all events. For example, the following commands register interest in events:

```
RegisterEventInterest("Smpsrv.Document","/user/rdbms/fault/event1","system1");
RegisterEventInterest("Smpsrv.Document", "*", "*");
```

This service must implement the `EventNotification` API to receive notices of triggered events. You must register an event set with the Console Event Management system before calling `RegisterEventInterest`. For information on events available with Enterprise Manager, see the "Event Management System" chapter of the *Oracle Enterprise Manager Administrator's Guide*.

`RegisterEventInterest` does not return errors when errors occur.

VoxEventNotifyUnpacker

The VoxEventNotifyUnpacker methods are:

- GetDate
- GetEventName
- GetFinalResult
- GetNodeName
- GetObjectName
- GetSeverity

GetDate

Purpose `GetDate` returns the date and time that the event was triggered.

Syntax `const CString& GetDate();`

Comments None.

GetEventName

Purpose `GetEventName` returns the name of the event.

Syntax `const CString& GetEventName();`

Comments None.

GetFinalResult

Purpose `GetFinalResult` returns the result string from Tcl event script.

Syntax `const CString& GetFinalResult();`

Comments None.

GetNodeName

Purpose `GetNodeName` returns the node name where the event occurred.

Syntax `const CString& GetNodeName();`

Comments None.

GetObjectName

Purpose `GetObjectName` returns the name of service object.

Syntax `const CString& GetObjectName();`

Comments None.

GetSeverity

Purpose `GetSeverity` returns the severity of the event.

Syntax `int GetSeverity();`

Comments The severity of the event is `-1` (clear), `1` (warning), or `2` (alert)

9

Jobs and Events Scripts

This chapter describes jobs and event scripts. Topics include:

- Scripting Language
- Server Message and Error Information
- Use of Tcl with the Intelligent Agent
- NLS Issues and Error Messages
- OraTcl Functions and Parameters

Scripting Language

The Tcl Language with OraTcl extensions is used to write the job and events scripts. Tcl is used for the scripts because it fulfills the necessary requirements, such as:

- Host system access for handling with files and devices, launching programs, and executing operating system functions.
- SQL and PL/SQL functions for accessing the RDBMS.
- RDBMS administration functions.
- SNMP accessing, both for the database MIB variables that the agent itself supports, and for external MIBs, like the host's or other SNMP-enabled services.
- Communication with the Oracle Intelligent Agent and other Oracle software, such as Oracle Trace.
- A syntax for describing job and event scripts that:
  - Can be used to drive the user interface.
  - Provide information on the nature of the job or event, and any input or output.
  - Allow access to the Oracle message file system for NLS support.

Tcl Language Description

Tcl originated with Dr. John Ousterhout from the University of California, Berkeley, California. Tcl, current release version 7.5, stands for *Tool Command Language*.

Tcl is both a language and a library. Tcl is a simple textual language that is intended primarily for issuing commands to interactive programs, such as text editors, debuggers, illustrators, and shells. Tcl has a simple syntax and is programmable. Tcl users can write command procedures to provide more powerful commands than those in the built-in set.

Tcl is also a library package that can be embedded in application programs. The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in functions, and procedures that allow each application to extend Tcl with additional commands specific to that application. The application program generates Tcl commands and passes them to the Tcl parser for execution. Commands may be generated by reading characters from an input source, or by associating command strings with elements of the application's user interface, such as menu entries, buttons, or keystrokes. When the Tcl library receives commands it parses them into component fields and executes built-in commands directly. For

commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases commands will invoke recursive invocations of the Tcl interpreter by passing in additional strings to execute. Procedures, looping commands, and conditional commands all work in this way.

An application program gains several advantages by using Tcl for its command language.

- Tcl provides a standard syntax. After you learn Tcl, you are able to issue commands easily to any Tcl-based application.
- Tcl provides programmability. All a Tcl application needs to do is to implement a few application-specific low-level commands. Tcl provides many utility commands plus a general programming interface for building up complex command procedures. By using Tcl, applications do not need to re-implement these features.
- Extensions to Tcl provide mechanisms for communicating between applications by sending Tcl commands back and forth. The common Tcl language framework makes it easier for applications to communicate.

Tcl was designed with the philosophy that one should actually use two or more languages when designing large software systems. One for manipulating complex internal data structures, or where performance is key, and another, such as Tcl, for writing small scripts that tie together the c programming pieces and provide hooks for others to extend. For the Tcl scripts, ease of learning, ease of programming and ease of integrating are more important than performance or facilities for complex data structures and algorithms. Tcl was designed to make it easy to drop into a lower language when you come across tasks that make more sense at a lower level. In this way, the basic core functionality can remain small and one need only bring along pieces that one particular wants or needs. For more information on Tcl/Tk, access the following web sites:

- http://sunscript.sun.com/
- http://www.neosoft.com/tcl
- ftp://ftp.smli.com/pub/tcl/

Note: World Wide Web site locations often change and the addresses may not be available in the future.

OraTcl Description

Agent jobs and event scripts require both host system access for handling files and devices, launching programs, executing operating system functions, and accessing Oracle databases. OraTcl was developed to extend Tcl for Oracle usage and SNMP accessing. The categories of OraTcl functions are:

- SQL and PL/SQL functions
- RDBMS administration functions
- SNMP accessing
- Communication with the intelligent agent and other Oracle software
- Character set conversion and error handling verbs
- General purpose utility functions

For descriptions of the OraTcl functions and variables, see *OraTcl Functions and Parameters* on page 9-12.

Example: OraTcl Script

The following example illustrates the basic use of OraTcl.

```
#!/usr/local/bin/Tcl -f
#
# monthly_pay.Tcl
#
# usage: monthly_pay.Tcl [connect_string]
#  or    Tcl -f monthly_pay.Tcl [connect_string]
#
# sample program for OraTcl
# Tom Poindexter
#
# example of sql, pl/sql, multiple cursors
# uses Oracle demo table SCOTT.EMP
# uses id/pass from command line,
# or "scott/tiger" if not specified
#
# this example does not illustrate efficient sql!
# a simple report is produced of the monthly payroll
# for each jobclass
#
global oramsg
set find_jobs_sql { select distinct job from SCOTT.EMP }
set monthly_pay_pl {
```

```
begin
  select sum(sal) into :monthly
 from SCOTT.EMP
 where job like :jobclass;
end;
}
set idpass $argv
if {[string length $idpass] == 0} {
  set idpass "scott/tiger"
}
set lda [oralogon $idpass]
set cur1 [oraopen $lda]
set cur2 [oraopen $lda]
orasql $cur1 $find_jobs_sql
set job [orafetch $cur1]
while {$oramsg(rc) == 0} {
  oraplexec $cur2 $monthly_pay_pl :monthly "" :jobclass "$job"
  set total_for_job [lindex [orafetch $cur2] 0]
  puts stdout "Total monthly salary for job class $job = \$ $total_for_job"
  set job [orafetch $cur1]
}
oraclose $cur1
oraclose $cur2
oralogoff $lda
exit
```

Server Message and Error Information

OraTcl creates and maintains a Tcl global array `oramsg` to provide feedback of Oracle server messages. `oramsg` is also used to communicate with the OraTcl interface routines to specify NULL return values and LONG limits. In all cases except for `NULLVALUE` and `MAXLONG`, each element is reset to NULL upon invocation of any OraTcl command, and any element affected by the command is set. The `oramsg` array is shared among all open OraTcl handles.

Note: `oramsg` should be defined with the global statement in any Tcl procedure that needs it.

`oramsg` Elements

The following are `oramsg` elements.

oramsg (agent_characterset)
The character set of the agent, such as US7ASCII. This is used with the `convertin` and `convertout` verbs to convert character sets. See *convertin* on page 9-14 and *convertout* on page 9-15.

oramsg (db_characterset)
The character set of the database, such as US7ASCII. This is used with the `convertin` and `convertout` verbs to convert character sets. See *convertin* on page 9-14 and *convertout* on page 9-15.

oramsg (collengths)
A Tcl list of the lengths of the columns returned by `oracols`. `collengths` is only set by `oracols`.

oramsg (colprecs)
A Tcl list of the precision of the numeric columns returned by `oracols`. `colprecs` is only set by `oracols`. For non-numeric columns, the list entry is a null string.

oramsg (colscales)
A Tcl list of the scale of the numeric columns returned by `oracols`. `Colscales` is only set by `oracols`. For non-numeric columns, the list entry is a null string.

oramsg (coltypes)
A Tcl list of the types of the columns returned by `oracols`. coltypes is only set by `oracols`. Possible types returned are: CHAR, VARCHAR2 (Version 7), NUMBER,

LONG, rowid, DATE, RAW, LONG_RAW, MLSLABEL, RAW_MLSLABEL, or unknown.

oramsg (errortxt)
The message text associated with `rc`. Because the `oraplexec` function may invoke several SQL statements, there is a possibility that several messages may be received from the server.

oramsg (handle)
Indicates the handle of the last OraTcl function. The handle, a mapping in memory used to track commands, is set on every OraTcl command except where an invalid handle is used.

oramsg (jobid)
The job Id of the current job. Defined for job scripts only.

oramsg (language)
The NLS language of the Console, such as AMERICAN_AMERICA.US7ASCII.

oramsg (maxlong)
Can be set by the programmer to limit the amount of `LONG` or `LONG RAW` data returned by `orafetch`. The default is 32K Bytes. The maximum is 64K (Version 6) or 2147483647 (Version 7) bytes. Any value less than or equal to zero is ignored. Any change to maxlong becomes effective on the next call to `orasql`. See notes on `MAXLONG` usage with `orafetch`.

oramsg (nullvalue)
Can be set by the programmer to indicate the string value returned for any NULL result. Setting `oramsg`(nullvalue) to DEFAULT will return 0 for numeric null data types, such as INTEGER, FLOAT, and MONEY, and a NULL string for all other data types. `NULLVALUE` is initially set to `default`.

oramsg (ocifunc)
The number OCI code of the last OCI function called by OraTcl. See the *Programmer's Guide to the Oracle Call Interface* for descriptions.

oramsg (oraobject)
Contains the object upon which this script is acting. Defined for event scripts only.

oramsg (orahome)
The ORACLE_HOME directory.

oramsg (oraindex)
A Tcl list of the SNMP index values from the `snmp.ora` configuration file.

oramsg (orainput)
A Tcl list that contains the names of the job's input files. Probably most jobs will not need input files, but a job which invokes SQL*Plus with a SQL script, or Export with a specification file, would use this feature. Defined for job scripts only.

oramsg (rc)
Indicates the results of the last SQL command and subsequent `orafetch` processing. `rc` is set by `orasql`, `orafetch`, `oraplexec`, and is the numeric return code from the last OCI library function called by an OraTcl command.

See the *Oracle Error Messages and Codes Manual* for detailed information.Typical values are listed in Table 9–1, "Error Messages".

Table 9–1 Error Messages

Error	Meaning
0000	Function completed normally, without error.
0900 - 0999	Invalid SQL statement, invalid sql statements, missing key-words, invalid column names, etc.
1000 - 1099	Program interface error. For example, no sql statement, logon denied, or insufficient privileges.
1400 - 1499	Execution errors or feedback.
1403	End of data was reached on an `orafetch` command.
1406	A column fetched by `orafetch` was truncated. Can occur when fetching a LONG or LONG RAW, and the maxlong value is smaller than the actual data size.

oramsg (rows)
The number of rows affected by an insert, update, or delete in an `orasql` command, or the cumulative number of rows fetched by `orafetch`.

oramsg (sqlfunc)
The numeric OCI code of the last SQL function performed. See the *Programmer's Guide to the Oracle Call Interface* for descriptions.

oramsg (starttime)
The time at which the job was scheduled to be started. Defined for jobs only.

Use of Tcl with the Intelligent Agent

Tcl scripts are used by the intelligent agent for jobs and events. While both are Tcl scripts, they are distinct in the agent and in the user interface.

Jobs are scripts scheduled to run once or multiple times. They typically cause side-effects, such as starting up a database, performing a backup, or sending output to the screen via the puts command, and can potentially have long execution times. Jobs can have output files and input files, such as a SQL script, while event scripts do not. Note that output files on Unix, DOS, or OS/2 are `stdout` redirected.

Event scripts, on the other hand, are used uniquely for detecting exceptions. A Tcl event script can monitor databases, host systems, or SQL*Net services by using a variety of means. If the script determines that a certain condition has occurred, it can send a return code to the agent that states the severity of the event. Event scripts tend to run more frequently than jobs and so they are expected to have relatively short execution times. Also, it is assumed that event scripts do not cause any side effects.

While both jobs and events use Tcl to accomplish their tasks, they are very different in nature and as such have different execution environments. Specifically, on UNIX systems, jobs are forked into a separate process, while events are usually executed in-line with the agent code.

The Tcl interpreter state is saved between executions and the value of Tcl global variables is preserved, for inline event scripts only, to give the illusion of a virtual process. This allows an event script to maintain a history so that the event does not get raised over and over again. For example, after you have notified the console that a value has gone above 90, you can refrain from notifying it again until the value goes below 80 and then back above 90. Database connections using the `oralogon` function are cached across all inline event scripts, so that repeated event scripts that use the same connect string can utilize the same connection.

Not all commands and global variables are available to both jobs and events. Jobs will not have the `oraobject` global variable that tells an event what service it is running against. Events will not have the `orainput` global that jobs use for SQL*Plus scripts.

NLS Issues and Error Messages

When a user registers for an event or schedules a job, the user's language preference is available to the agent. There is a special remote procedure call which reports the language and current address of each console user. The agent proceeds to issue an ALTER SESSION command to the specified language every time the `oralogon` function is called. This means that any subsequent messages or output coming from the Oracle server will be in the user's language. In addition, character set conversion is explicitly not done on the agent, so that the Console can do it on the user's side.

If an event script or a job script fails execution, an error message is sent back to the Console in the user's language. Typically this will be an Oracle message returned by one of the Oracle Tcl extensions, if the verb was given inadequate parameters. For example `oralogon` might return the error: "ERROR: ORA-01017: invalid username/password; logon denied" if it is given an incorrect connect string. However, the error message could also be a Tcl specific message, such as: "ERROR: Tcl-00456: division by zero error", which will be stored in a message file and thus can be returned in the user's preferred language. The default language used by the agent will be American English if no user language preference is specified or if an error message text does not exist in the user's language.

OraTcl Functions and Parameters

This section lists the OraTcl functions and parameters. Functions or other words that appear in OraTcl syntax are shown in this font: `function`. Parameters in square brackets '`[option]`' are optional, and the '|' character means 'or'. All parameters are passed into the functions and are IN mode.

- SQL and PL/SQL functions

oraautocom	*oracancel*	*oraclose*	*oracols*	*oracommit*
orafetch	*oralogoff*	*oralogon*	*oraopen*	*oraplexec*
orareadlong	*oraroll*	*orasql*	*orawritelong*	

- RDBMS administration functions

orastart	*orastop*

- SNMP accessing functions

oradbsnmp	*orasnmp*

- Communication with the Intelligent Agent and other Oracle software functions

orafail	*oragetfile*	*orainfo*	*orajobstat*	*orareportevent*

- Character set conversion and error handling functions

convertin	*convertout*	*msgtxt*	*msgtxt1*

- General purpose utility functions

catfile	*concatname*	*diskusage*	*echofile*	*export*
import	*loader*	*mvfile*	*orasleep*	*oratime*
rmfile	*tempdir*	*tempfile*		

Common Parameters

The following parameters are used in multiple OraTcl functions and the descriptions are provided in this section.

column
The column name that is the LONG or LONG RAW column.

connect_string
A valid Oracle database connect string, in one of the forms:

```
name | name/password | name@n:dbname | name/password@n:dbname
```

destaddress
`destaddress` is the destination address of the agent.

filename
The name of the file that contains the LONG or LONG RAW data to write into the column or the name of the file in which to write the LONG or LONG RAW data.

logon-handle
A valid *cursor-handle* previously opened with `oraopen`. The handle is a mapping in memory used to track functions.

rowid
The Oracle database `rowid` of an existing row, and must be in the format of an Oracle `rowid` datatype.

table
The Oracle database table name that contains the row and column.

catfile

Purpose This function returns the contents of a file.

Syntax `catfile filename`

Parameters **filename**
The file that you want to display.

Example `catfile /tmp/files1 or c:/orant/sysman/admin/vobmgr.log`

concatname

Purpose This function returns the full pathname for a file given a list of file name components.

Syntax `concatname components`

Parameters **components**
A list containing the filename and each directory name where the file is located.

Example `concatname [list $oramsg(orahome) network agent]`

convertin

Purpose This function converts the parameter string from the client's (Console) character set to the destination character set. The function returns the converted string.

Syntax `convertin dest_characterset string`

Parameters **dest_characterset**
Destination character set. For database specific jobs or events, use $oramsg(db_characterset). For node specific jobs or events, use $oramsg(agent_characterset). See *oramsg Elements* on page 9-6.

string
The string that is converted.

Comments The client and the agent node may use different languages or character sets. It is the responsibility of the Tcl script developer to perform the character set conversion. In general, all the job or event input parameters should be converted unless they are guaranteed to be ASCII.

convertout

Purpose This function converts the parameter string from the destination character set to the client's (Console) character set. The function returns the converted string.

Syntax `convertout dest_characterset string`

Parameters **dest_characterset**
Destination character set. For database specific jobs or events, use $oramsg(db_characterset). For node specific jobs or events, use $oramsg(agent_characterset). See *oramsg Elements* on page 9-6.

string
The string that is converted.

Comments The client and the agent node may use different languages or character sets. It is the Tcl script developers' responsibility to perform the character set conversion. In general all the job or event output should be converted unless they are guaranteed to be ASCII.

diskusage

Purpose This function returns disk usage information on a list of files. The output is four lists: file systems, total space, available space, and mount points.

Syntax `diskusage files`

Parameters **files**
A list of file names. You can omit the filenames to display information on the entire file system.

Example `diskusage [list /tmp]`

echofile

Purpose This function writes a string to a file.

Syntax `echofile string filename`

Parameters **string**
The string you want to write to the file.

filename
The name of the file where you want to store the string.

Example `echofile asdf /tmp/temp`

export

Purpose This function executes the Oracle Export database tool.

Syntax `export arguments`

Parameters **arguments**
These are the command-line arguments that are used by the Export tool.

Comments For information on Export, see the *Oracle7 Server Utilities User's Guide.*

import

Purpose This function executes the Oracle Import database tool.

Syntax `import arguments`

Parameters **arguments**
These are the command-line arguments that are used by the Import tool.

Comments For information on Import, see the *Oracle7 Server Utilities User's Guide.*

loader

Purpose This function executes the Oracle SQL*Loader database tool.

Syntax `loader arguments`

Parameters **arguments**
These are the command-line arguments that are used by the SQL*Loader tool.

Comments For information on SQL*Loader, see the *Oracle7 Server Utilities User's Guide*.

msgtxt

Purpose This function returns message text in the client's (Console) language for the given product name, facility and message number. The output is in the format of "FACILITY-ERROR : MESSAGE TEXT".

Syntax `msgtxt product facility error_no`

Parameters **product**
Product name. For example, `rdbms`.

facility
Facility name. For example, `ora`.

error_no
Error number. For example, `1101`.

Comments This function is used to put out error messages in the job output file. The message will be displayed in the client's (Console) language.

msgtxt1

Purpose This function returns a message in the client's (Console) language for the given product name, facility and message number. The output is in the format of "MESSAGE TEXT".

Syntax `msgtxt1 product facility error_no`

Parameters **product**
Product name. For example, `rdbms`.

facility
Facility name. For example, `ora`.

error_no
Error number. For example, `1101`.

Comments This function is used to put out confirmation messages in the job output file. The message will be displayed in the client's (Console) language.

mvfile

Purpose This function moves a file to a different location / name.

Syntax `mvfile filename destination`

Parameters **filename**
The name of the file you want to move or rename.

destination
The new destination / name.

Comments None

oraautocom

Purpose This function enables or disables automatic commit of SQL data manipulation statements using a cursor opened through the connection specified by `logon-handle`.

Syntax `oraautocom logon-handle {on | off}`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oraautocom` raises a Tcl error if the `logon-handle` specified is not open.

Either `on` or `off` must be specified. The automatic commit feature defaults to off.

oracancel

Purpose This function cancels any pending results from a prior `orasql` function that use a cursor opened through the connection specified by `logon-handle`.

Syntax `oracancel logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oracancel` raises a Tcl error if the `logon-handle` specified is not open.

oraclose

Purpose This function closes the cursor associated with `logon-handle`.

Syntax `oraclose logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oraclose` raises a Tcl error if the `logon-handle` specified is not open.

oracols

Purpose This function returns the names of the columns from the last `orasql`, `orafetch`, or `oraplexec` function as a Tcl list. `oracols` may be used after `oraplexec`, in which case the bound variable names are returned.

Syntax `oracols logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oracols` raises a Tcl error if the `logon-handle` specified is not open.

The `oramsg` array index `collengths` is set to a Tcl list corresponding to the lengths of the columns; index `coltypes` is set to a Tcl list corresponding to the types of the columns; index `colprecs` is set to a Tcl list corresponding to the precision of the numeric columns, other corresponding non-numeric columns are a null string (Version 7 only); index `colscales` is set to a Tcl list corresponding to the scale of the numeric columns, other corresponding non-numeric columns are a null string (Version 7 only).

oracommit

Purpose This function commits any pending transactions from prior `orasql` functions using a cursor opened with the connection specified by `logon-handle`.

Syntax `oracommit logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oracommit` raises a Tcl error if the logon handle specified is not open.

oradbsnmp

Purpose This function retrieves SNMP MIB values.

Syntax `oradbsnmp get | getnext object_Id`

Parameters **object_Id**
`object_Id` can be either an actual MIB object Id, such as "1.3.6.1.2.1.1.1.0", or an object name with a possible index attached to it, such as "sysDescr" or "sysDescr.0".

Comments `oradbsnmp` is a function for retrieving SNMP MIB values maintained by the agent, such as the RDBMS public MIB or the Oracle RDBMS private MIB. It does not write to the well-known UDP port for SNMP and obtains its values directly from the agent's internal data structures. It works if the host does not have an SNMP master agent running on it. See *orasnmp* on page 9-31 for more details on what `get` and `getnext` do. There are several reasons why `oradbsnmp` should be used instead of fetching the values from V$ tables with SQL commands:

- The agent maintains a cache of MIB values fetched from the V$ tables to avoid burdening the RDBMS excessively. `oradbsnmp` is often faster than SQL and imposes less overhead on the system.
- When SGA access is implemented, it will be transparent to this function, for those MIB variables that are fetched directly from the SGA.
- In the case of `getnext`, the next `object_id` is the next `object_id` within the private and public RDBMS MIBs, and not one of another MIB. It is impossible to retrieve system-specific information using this function; use `orasnmp`.

orafail

Purpose This function forces a Tcl script to fail.

Syntax `orafail errormsg`

Parameters **errormsg**
`errormsg` can either be a quoted string of text or a string of the form: FAC-XXXXX where XXXXX is an Oracle message number for the given facility, such as VOC-99999.

Comments The error message will be used for display purposes on the client side.

orafetch

Purpose This function returns the next row from the last SQL statement executed with `orasql` as a Tcl list.

Syntax `orafetch logon-handle [commands]`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

commands
The optional commands allows `orafetch` to repeatedly fetch rows and execute commands for each row.

Comments `orafetch` raises a Tcl error if the `logon-handle` specified is not open.

All returned columns are converted to character strings. A null string is returned if there are no more rows in the current set of results. The Tcl list that is returned by `orafetch` contains the values of the selected columns in the order specified by select.

Substitutions are made on commands before passing it to Tcl_Eval() for each row. `orafetch` interprets @n in commands as a result column specification. For example, @1, @2, @3 refer to the first, second, and third columns in the result. @0 refers to the entire result row, as a Tcl list. Substitution columns may appear in any order, or more than once in the same command. Substituted columns are inserted into the commands string as proper list elements. For example, one space will be added before and after the substitution and column values with embedded spaces are enclosed by {} if needed.

A Tcl error is raised if a column substitution number is greater than the number of columns in the results. If the commands execute a break, `orafetch` execution is interrupted and returns with Tcl_OK. Remaining rows may be fetched with a subsequent `orafetch` function. If the commands execute return or continue, the remaining commands are skipped and `orafetch` execution continues with the next row. `orafetch` will raise a Tcl error if the commands return an error. Commands should be enclosed in "" or {}.

OraTcl performs conversions for all data types. Raw data is returned as a hexadecimal string, without a leading "0x". Use the SQL functions to force a specific conversion.

The `oramsg` array index `rc` is set with the return code of the fetch. 0 indicates the row was fetched successfully; 1403 indicates the end of data was reached. The index of rows is set to the cumulative number of rows fetched so far.

The `oramsg` array index maxlong limits the amount of long or long raw data returned for each column returned. The default is 32768 bytes. The `oramsg` array index nullvalue can be set to specify the value returned when a column is null. The default is "0" for numeric data, and "" for other datatypes.

oragetfile

Purpose This function is used by jobs to copy a remote file into a local file.

Syntax `oragetfile destaddress remotefile localfile [BIN]`

Parameters

destaddress
See *Common Parameters* on page 9-13.

remotefile
`remotefile` is the name of the file that is the source of the copy.

localfile
`localfile` is the name of the file that is the target of the copy.

Comments `oragetfile` fetches the file `remotefile` into the local file `localfile` from the agent at `destaddress`. If the `BIN` argument is specified, the file is transferred in binary mode.

`destaddress` may be obtained from the `orainfo` function. Note that the address provided must be the spawn address of the agent, the special address on which it listens for file transfer requests, and not the normal address used for all other RPCs.

Additional Information: For more information on the address of an intelligent agent, see the chapter on configuring the agent in the *Oracle Enterprise Manager Installation Guide*.

orainfo

Purpose This function is used by jobs to get configuration information.

Syntax `orainfo destaddress`

Parameters

destaddress
See *Common Parameters* on page 9-13.

Comments `orainfo` fetches agent configuration information from the agent at `destaddress`. If `destaddress` is not present, then it is fetched from the agent on the local machine. The agent configuration is a Tcl list, as follows:

- A list of databases monitored by this agent. The list includes the database name, `ORACLE_HOME`, and SID for each database.
- The agent's normal RPC address, a tnsnames (TNS) string.
- The agent's file transfer address, a TNS string.

orajobstat

Purpose This function is used by a job to send intermediate output back to the Console.

Syntax `orajobstat destaddress string`

Parameters

destaddress
See *Common Parameters* on page 9-13.

string
`string` can either be a quoted string of text or a string of the form: FAC-XXXXX where XXXXX is an Oracle message number for the given facility, such as VOC-99999. The string is used for display on the client side.

Comments `destaddress` is the address of the agent, not the daemon. This function is issued from a job process, not from within an agent process. The agent's address can be obtained with `orainfo`.

oralogoff

Purpose This function logs off from the Oracle server connection associated with `logon-handle`.

Syntax `oralogoff logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oralogoff` raises a Tcl error if the logon handle specified is not open. `oralogoff` returns a null string.

oralogon

Purpose This function connects to an Oracle server using `connect_string`.

Syntax `oralogon connect_string`

Parameters

connect_string
See *Common Parameters* on page 9-13.

Comments A `logon-handle` is returned and should be used for all other OraTcl functions using this connection that require a `logon-handle`. Multiple connections to the same or different servers are allowed, up to a maximum of six total connections.

Additional Information: The connection limit is covered in the operating system-specific notes. When `oralogon` is used in an event script, it benefits from the connection cache. It will usually be able to reuse the connections opened by other event scripts against the same database. See *NLS Issues and Error Messages* on page 9-11 for details. `oralogon` raises a Tcl error if the connection is not made for any reason, such as login incorrect or network unavailable. If connect_string does not include a database specification, the value of the environment variable `ORACLE_SID` is used as the server.

oraopen

Purpose This function opens an SQL cursor to the server. `oraopen` returns a cursor to be used on subsequent OraTcl functions that require a `logon-handle`.

Syntax `oraopen logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oraopen` raises a Tcl error if the `logon-handle` specified is not open. Multiple cursors can be opened through the same or different logon handles, up to a maximum of 25 total cursors.

oraplexec

Purpose This function executes an anonymous PL block, optionally binding values to PL/SQL variables.

Syntax `oraplexec logon-handle pl_block [:varname value ...]`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

pl_block
`pl_block` may either be a complete PL/SQL procedure or a call to a stored procedure coded as an anonymous PL/SQL block.

:varname value
`:varname value` are optional pairs.

Comments `oraplexec` raises a Tcl error if the `logon-handle` specified is not open, or if the PL/SQL block is in error. `oraplexec` returns the contents of each :varname as a Tcl list upon the termination of PL/SQL block.

Optional :varname value pairs may follow `pl_block`. Varnames must be preceded by a colon, and match the substitution names used in the procedure. Any `:varname` that is not matched with a value is ignored. If a `:varname` is used for output, the value should be coded as a null string, "".

The `oramsg` array index rc contains the return code from the stored procedure.

orareadlong

Purpose This function reads the contents of a LONG or LONG RAW column and write results into a file.

Syntax `orareadlong logon-handle rowid table column filename`

Parameters

logon-handle rowid table column filename
See *Common Parameters* on page 9-13.

Comments orareadlong returns a decimal number upon successful completion of the number of bytes read from the LONG column.

orareadlong raises a Tcl error if the logon-handle specified is not open, or if rowid, table, or column are invalid, or if the row does not exist.

orareadlong composes and executes an SQL select statement based on the table, column, and rowid. A properly formatted Rowid may be obtained through a prior execution of orasql, such as "SELECT rowid FROM table WHERE ...".

orareportevent

Purpose This function is used by jobs to report an unsolicited event to the agent and Event Management system in the Console. The oemevent executable can also be used.

Syntax orareportevent eventname object severity message [results]

Parameters eventname
eventname is the name of the event. This is the four-part name of the event in the form:

```
/vendor/product/category/name
```

You can enter any character strings but all four parts and the forward slashes (/) are required. If internal is used in the third position, the event is not sent to the Outstanding Events window in the Console.

The first two levels of name have special significance and have many predefined strings that Oracle script writers must use:

- Level one is the definer of this script, typically the integrating company name such as oracle, or user for unspecified customers.
- Level two is the name of the product to which this script is related, for example rdbms, office, agent, osgeneric, sqlnet, or hpux. All Oracle services have defined names which Oracle script writers must use.

The eventname is assumed to be in 7-bit ASCII, so that it never changes regardless of platform or language. See eventdef.tcl in the ORACLE_HOME\net8\admin directory (Oracle Enterprise Manager release 1.5.0 on a Windows NT platform) for a list of defined event names.

Note: The actual event script name may be shortened, upper-cased, or manipulated in other ways to make it a legal, unique filename on a given platform. The format is operating system-specific. For example, `/oracle/rdbms/security/SecurityError` can be stored as `$oracle_home/network/agent/events/oracle/rdbms/security/securityerror.tcl` on a Unix system.

object
`object` is the name of the object that the event is monitoring, such as the database or service name listed in the `snmp.visibleservices` parameter in the `snmp.ora` file, or `$oramsg(nodename)`.

severity
`severity` is the level of severity of the event. For `orareportevent`, the value is `1` (warning), `2` (alert), or `-1` (clear). For `oemevent`, this is the literal text string `alert`, `warning`, or `clear`.

message
`message` is a quoted text string that is displayed in the Console, such as "File not found."

[results]
`results` is any results that may occur from the event. This is a Tcl list with the specific results for the event, such as the tablespace in error or the user who had a security violation.

Comments This is the method for any job to report an unsolicited event to the agent, and back to the Console. For more information, see *Server-Side Integration* on page 8-4. For information on the Event Management system, see the *Oracle Enterprise Manager Administrator's Guide*.

oraroll

Purpose This function rolls back any pending transactions from prior `orasql` functions that use a cursor opened through the connection specified by `logon-handle`.

Syntax `oraroll logon-handle`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

Comments `oraroll` raises a Tcl error if the logon handle specified is not open.

orasleep

Purpose This function causes the Tcl script to pause for a number of seconds.

Syntax `orasleep seconds`

Parameters **seconds**

Comments `orasleep` calls slcsleep() for the required number of seconds. There is no default, minimum, or maximum value.

orasnmp

Purpose This function performs either an SNMP `get` or `getnext` operation on the object specified by `object_id`.

Syntax `orasnmp get | getnext object_Id`

Parameters **object_Id**
The `object_Id` can be either an actual MIB object Id, such as "1.3.6.1.2.1.1.1.0", or an object name with an index attached to it, such as "sysDescr" or "sysDescr.0".

Comments Object names come from MIB text files. A full network manager, such as OpenView, has a MIB compiler that accepts MIB files and parses the ASN.1, creating a database of all objects in all the MIBs. The agent needs to be simpler. There is a standard configuration directory which contains one or more two-column ASCII files of the format:

```
"rdbmsDbPrivateMibOID",   "1.3.6.1.2.1.39.1.1.1.2",
"rdbmsDbVendorName",      "1.3.6.1.2.1.39.1.1.1.3",
"rdbmsDbName",            "1.3.6.1.2.1.39.1.1.1.4",
"rdbmsDbContact",         "1.3.6.1.2.1.39.1.1.1.5",
....
```

The Tcl interpreter reads these files and does a binary search on them at runtime to resolve an object name to an `object_Id`.

The index values to use for Oracle services are configured via the `snmp.ora` file. These indices can also be obtained from the `oraindex` global variable. See *Server Message and Error Information* on page 9-6.

The result of `orasnmp` is a Tcl list of the form:

```
{object_id   value}
```

where `object_Id` is the object id associated with `value`. In the case of an `orasnmp get`, `object_Id` is the same as object, while for a `getnext`, it would be the next logical `object_Id`. It is assumed that the `orasnmp` operation applies to the local host only. The function actually sends out an SNMP query to the well-known SNMP port on the local host, so it is possible to query MIB variables other than Oracle's, such as those of the host or other applications that support SNMP. An SNMP master agent needs to be running on the local host for this function to work. See *oradbsnmp* on page 9-21 for an optimized way to retrieve the Oracle database MIB objects. If the master agent is not running, this function fails.

orasql

Purpose This function sends the Oracle SQL statement SQL statement to the server.

Syntax `orasql logon-handle sql_stmt`

Parameters

logon-handle
See *Common Parameters* on page 9-13.

sql_stmt
`sql_stmt` is a single, valid SQL statement.

Comments `logon-handle` must be a valid handle previously opened with `oraopen`. `orasql` raises a Tcl error if the `logon-handle` specified is not open, or if the SQL statement is syntactically incorrect.

`orasql` will return the numeric return code 0 on successful execution of the SQL statement. The `oramsg` array index rc is set with the return code; the rows index is set to the number of rows affected by the SQL statement in the case of insert, update, or delete. Only a single SQL statement may be specified in `sql_stmt`. `orafetch` allows retrieval of return rows generated. `orasql` performs an implicit `oracancel` if any results are still pending from the last execution of `orasql`.

Table inserts made with `orasql` should follow conversion rules in the Oracle SQL Reference manual.

orastart

Purpose This function starts an Oracle database instance.

Syntax `orastart connect_string [init_file] [SYSDBA|SYSOPER] [RESTRICT] [PARALLEL] [SHARED]`

Parameters

connect_string
See *Common Parameters* on page 9-13.

init_file
`init_file` is the path to the `init.ora` file to use.

Comments The default for `init_file` is:

```
ORACLE_HOME/dbs/init${ORACLE_SID}.ora
```

[SYSDBA | SYSOPER] are role flags for the user starting up the database. [RESTRICT] [PARALLEL] [SHARED] are database options. If [RESTRICT] is specified, database is started in restricted mode.

orastop

Purpose This function stops an Oracle database instance.

Syntax `orastop connect_string [SYSDBA|SYSOPER] [IMMEDIATE|ABORT]`

Parameters

connect_string
See *Common Parameters* on page 9-13.

Comments [SYSDBA | SYSOPER] are role flags for the user shutting down the database. [IMMEDIATE | ABORT] are the shutdown mode flags.

Note: Shutdown normal might be expected to fail every time, because the agent maintains its own connection to the database, but we send a special RPC to the agent when this is done, which causes it to disconnect from the database.

oratime

Purpose This function returns the current date and time.

Syntax `oratime`

Parameters None

Comments None

orawritelong

Purpose This function writes the contents of a file to a LONG or LONG RAW column.

Syntax `orawritelong logon-handle rowid table column filename`

Parameters

logon-handle rowid table column filename
See *Common Parameters* on page 9-13.

Comments `orawritelong` composes and executes an SQL update statement based on the table, column, and rowid. `orawritelong` returns a decimal number upon successful completion of the number of bytes written to the LONG column. A properly formatted ROWID may be obtained through a prior execution of the `orasql` function, such as "SELECT rowid FROM table WHERE ....".

`orawritelong` raises a Tcl error if the `logon-handle` specified is not open, or if rowid, table, or column are invalid, or if the row does not exist.

rmfile

Purpose This function removes a file.

Syntax `rmfile filename`

Parameters **filename**
This is the file that you want to remove.

Comments None

tempdir

Purpose This function returns a directory name that is used to store temporary files.

Syntax `tempdir`

Parameters **none**

Comments On Unix platforms, the `tempdir` is usually `/tmp`. On Windows NT systems, the tempdir is usually `\temp`.

tempfile

Purpose This function returns a temporary filename with the given extension. The filename is generated using the timestamp.

Syntax `tempfile extension`

Parameters **extension**
This is the file extension of the temporary file.

Example `tempfile dd`
returns the following on a Unix platform

```
/tmp/143153.dd
```

A

NLS Codes

The following are the codes used by National Language Support (NLS).

Table A–1 NLS Codes

Code	Language	Territory
US	American	America
AR	Arabic	UAE
PTB	Brazilian Portuguese	Brazil
BG	Bulgarian	Bulgaria
FRC	Canadian French	Canada
CA	Catalan	Catalonia
HR	Croatian	Croatia
CS	Czech	Czechoslovakia
DK	Danish	Denmark
NL	Dutch	The Netherlands
EG	Egyptian	Egypt
GB	English	United Kingdom
SF	Finnish	Finland
F	French	France
D	German	Germany
El	Greek	Greece

Table A–1 NLS Codes

Code	Language	Territory
IW	Hebrew	Israel
HU	Hungarian	Hungary
IS	Icelandic	Iceland
I	Italian	Italy
JA	Japanese	Japan
KO	Korean	Korea
LT	Lithuanian	Lithuania
ESM	Mexican Spanish	Mexico
N	Norwegian	Norway
PL	Polish	Poland
PT	Portuguese	Portugal
RO	Romanian	Romania
RU	Russian	CIS
ZHS	Simplified Chinese	China
SK	Slovak	Czechoslovakia
SL	Slovenian	Slovenia
E	Spanish	Spain
S	Swedish	Sweden
TH	Thai	Thailand
ZHT	Traditional Chinese	Taiwan
TR	Turkish	Turkey

Index

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W