TIBCO® Object Service Broker for Open Systems

External Environments

Software Release 6.0
July 2012

two-second advantage™

TIBCO®
The Power of Now®

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, The Power of Now, TIBCO Object Service Broker, and and TIBCO Service Gateway are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

The TIBCO Object Service Broker technologies described herein are protected under the following patent numbers:

Australia:	-	-	671137	671138	673682	646408
Canada:	2284250	-	-	2284245	2284248	2066724
Europe:	-	-	0588446	0588445	0588447	0489861
Japan:	-	-	-	-	-	2-513420
USA:	5584026	5586329	5586330	5594899	5596752	5682535

Contents

Preface

TIBCO® Object Service Broker is an application development environment and integration broker that bridges legacy and non-legacy applications and data.

This manual provides information on interfacing TIBCO Object Service Broker with the Open Systems operating environments. It also includes information on how to use C language programs to access TIBCO Object Service Broker data and how to access C language programs from within TIBCO Object Service Broker.

Topics

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Object Service Broker Documentation

The following documents form the TIBCO Object Service Broker documentation set:

Fundamental Information

The following manuals provide fundamental information about TIBCO Object Service Broker:

- *TIBCO Object Service Broker Getting Started* Provides the basic concepts and principles of TIBCO Object Service Broker and introduces its components and capabilities. It also describes how to use the default developer's workbench and includes a basic tutorial of how to build an application using the product. A product glossary is also included in the manual.
- *TIBCO Object Service Broker Messages with Identifiers* Provides a listing of the TIBCO Object Service Broker messages that are issued with alphanumeric identifiers. The description of each message includes the source and explanation of the message and recommended action to take.
- *TIBCO Object Service Broker Messages without Identifiers* Provides a listing of the TIBCO Object Service Broker messages that are issued without a message identifier. These messages use the percent symbol (%) or the number symbol (#) to represent such variable information as a rules name or the number of occurrences in a table. The description of each message includes the source and explanation of the message and recommended action to take.
- *TIBCO Object Service Broker Quick Reference* Presents summary information for use in the TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Shareable Tools* Lists and describes the TIBCO Object Service Broker shareable tools. Shareable tools are programs supplied with TIBCO Object Service Broker that facilitate rules language programming and application development.
- *TIBCO Object Service Broker Release Notes* Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

Application Development and Management

The following manuals provide information about application development and management:

- *TIBCO Object Service Broker Application Administration* Provides information required to administer the TIBCO Object Service Broker application development environment. It describes how to use the administrator's workbench, set up the development environment, and optimize access to the database. It also describes how to manage the Pagestore, which is the native TIBCO Object Service Broker data store.
- *TIBCO Object Service Broker Managing Data* Describes how to define, manipulate, and manage data required for a TIBCO Object Service Broker application.
- *TIBCO Object Service Broker Managing External Data* Describes the TIBCO Object Service Broker interface to external files (not data in external databases) and describes how to define TIBCO Object Service Broker tables based on these files and how to access their data.
- *TIBCO Object Service Broker National Language Support* Provides information about implementing the National Language Support in a TIBCO Object Service Broker environment.
- *TIBCO Object Service Broker Object Integration Gateway* Provides information about installing and using the Object Integration Gateway which is the interface for TIBCO Object Service Broker to XML, J2EE, .NET and COM.
- *TIBCO Object Service Broker for Open Systems External Environments* Provides information on interfacing TIBCO Object Service Broker with the Windows and Solaris environments. It includes how to use SDK (C/C++) and SDK (Java) to access TIBCO Object Service Broker data, how to interface to TIBCO Enterprise Messaging Service (EMS), how to use the TIBCO Service Gateway for WMQ, how to use the Adapter for JDBC-ODBC, and how to access programs written in external programming languages from within TIBCO Object Service Broker.
- *TIBCO Object Service Broker for z/OS External Environments* Provides information on interfacing TIBCO Object Service Broker to various external environments within a TIBCO Object Service Broker z/OS environment. It also includes information on how to access TIBCO Object Service Broker from different terminal managers, how to write programs in external programming languages to access TIBCO Object Service Broker data, how to interface to TIBCO Enterprise Messaging Service (EMS), how to use the TIBCO Service Gateway for WMQ, and how to access programs written in external programming languages from within TIBCO Object Service Broker.

- *TIBCO Object Service Broker Parameters* Lists the TIBCO Object Service Broker Execution Environment and Data Object Broker parameters and describes their usage.
- *TIBCO Object Service Broker Programming in Rules* Explains how to use the TIBCO Object Service Broker rules language to create and modify application code. The rules language is the programming language used to access the TIBCO Object Service Broker database and create applications. The manual also explains how to edit, execute, and debug rules.
- *TIBCO Object Service Broker Managing Deployment* Describes how to submit, maintain, and manage promotion requests in the TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Defining Reports* Explains how to create both simple and complex reports using the reporting tools provided with TIBCO Object Service Broker. It explains how to create reports with simple features using the Report Generator and how to create reports with more complex features using the Report Definer.
- *TIBCO Object Service Broker Managing Security* Describes how to set up, use, and administer the security required for an TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Defining Screens and Menus* Provides the basic information to define screens, screen tables, and menus using TIBCO Object Service Broker facilities.
- *TIBCO Service Gateway for Files SDK* Describes how to use the SDK provided with the TIBCO Service Gateway for Files to create applications to access Adabas, CA Datacom, and VSAM LDS data.

System Administration on the z/OS Platform

The following manuals describe system administration on the z/OS platform:

- *TIBCO Object Service Broker for z/OS Installing and Operating* Describes how to install, migrate, update, maintain, and operate TIBCO Object Service Broker in a z/OS environment. It also describes the Execution Environment and Data Object Broker parameters used by TIBCO Object Service Broker.
- *TIBCO Object Service Broker for z/OS Managing Backup and Recovery* Explains the backup and recovery features of OSB for z/OS. It describes the key components of TIBCO Object Service Broker systems and describes how you can back up your data and recover from errors. You can use this information, along with assistance from TIBCO Support, to develop the best customized solution for your unique backup and recovery requirements.

- *TIBCO Object Service Broker for z/OS Monitoring Performance* Explains how to obtain and analyze performance statistics using TIBCO Object Service Broker tools and SMF records
- *TIBCO Object Service Broker for z/OS Utilities* Contains an alphabetically ordered listing of TIBCO Object Service Broker utilities for z/OS systems. These are TIBCO Object Service Broker administrator utilities that are typically run with JCL.

System Administration on Open Systems

The following manuals describe system administration on open systems such as Windows or UNIX:

- *TIBCO Object Service Broker for Open Systems Installing and Operating* Describes how to install, migrate, update, maintain, and operate TIBCO Object Service Broker in Windows and Solaris environments.
- *TIBCO Object Service Broker for Open Systems Managing Backup and Recovery* Explains the backup and recovery features of TIBCO Object Service Broker for Open Systems. It describes the key components of a TIBCO Object Service Broker system and describes how to back up your data and recover from errors. Use this information to develop a customized solution for your unique backup and recovery requirements.
- *TIBCO Object Service Broker for Open Systems Utilities* Contains an alphabetically ordered listing of TIBCO Object Service Broker utilities for Windows and Solaris systems. These TIBCO Object Service Broker administrator utilities are typically executed from the command line.

External Database Gateways

The following manuals describe external database gateways:

- *TIBCO Service Gateway for DB2 Installing and Operating* Describes the TIBCO Object Service Broker interface to DB2 data. Using this interface, you can access external DB2 data and define TIBCO Object Service Broker tables based on this data.
- *TIBCO Service Gateway for IDMS/DB Installing and Operating* Describes the TIBCO Object Service Broker interface to CA-IDMS data. Using this interface, you can access external CA-IDMS data and define TIBCO Object Service Broker tables based on this data.
- *TIBCO Service Gateway for IMS/DB Installing and Operating* Describes the TIBCO Object Service Broker interface to IMS/DB and DB2 data. Using this interface, you can access external IMS data and define TIBCO Object Service Broker tables based on it.

- *TIBCO Service Gateway for ODBC and for Oracle Installing and Operating*
  Describes the TIBCO Object Service Broker ODBC Gateway and the TIBCO Object Service Broker Oracle Gateway interfaces to external DBMS data. Using this interface, you can access external DBMS data and define TIBCO Object Service Broker tables based on this data.

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
TIBCO_HOME *OSB_HOME*	By default, all TIBCO products are installed into a folder referenced in the documentation as *TIBCO_HOME*. On open systems, TIBCO Object Service Broker installs by default into a directory within *TIBCO_HOME*. This directory is referenced in documentation as *OSB_HOME*. The default value of *OSB_HOME* depends on the operating system. For example on Windows systems, the default value is `C:\tibco\OSB`. Similarly, all TIBCO Service Gateways on open systems install by default into a directory in *TIBCO_HOME*. For example on Windows systems, the default value is `C:\tibco\OSBgateways\6.0`. On z/OS, no default installation directories exist.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: `MyCommand [`**`enable`**` | disable]`
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO ActiveMatrix BusinessWorks Concepts*. • To introduce new terms For example: A portal page may contain several portlets. *Portlets* are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *PathName*

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2 Syntax Typographical Conventions

Convention	Use
`[ ]`	An optional item in a command or code syntax. For example: `MyCommand [optional_parameter] required_parameter`
`|`	A logical `OR` that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: `MyCommand para1 | param2 | param3`

Table 2 Syntax Typographical Conventions

Convention	Use

`{ }`	A logical group of items in a command. Other syntax notations may appear within each logical group.
	
	For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`.
	
	`MyCommand {param1 param2} | {param3 param4}`
	
	In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:
	
	`MyCommand {param1 | param2} {param3 | param4}`
	
	In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`.
	
	`MyCommand param1 [param2] {param3 | param4}`

Connecting with TIBCO Resources

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts, a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to http://www.tibcommunity.com.

How to Access All TIBCO Documentation

You can access TIBCO documentation here:

http://docs.tibco.com

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

Chapter 1 About the TIBCO Object Service Broker System

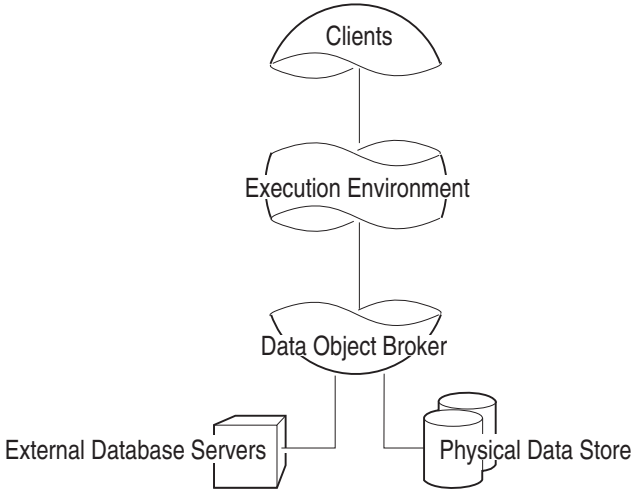
This chapter introduces the TIBCO Object Service Broker architecture within the client server model and describes how to access it from an external environment.

Topics

About TIBCO Object Service Broker Architecture

TIBCO Object Service Broker is a transactional processing environment consisting of an application development interface, a code execution environment, and a physical database.

These entities can all exist on one physical machine or they can be distributed across different machines. The following diagram illustrates this relationship:



Client Services Layer

The client services layer provides the interface between TIBCO Object Service Broker and its host operating environment, referred to as an *external environment* in this manual. This layer provides support for TIBCO Object Service Broker sessions running under, for example, SDK (C/C++), SDK (Java), Object Integration Gateway, Telnet 3270, or external routines. Every TIBCO Object Service Broker session that runs in an Execution Environment is started by a client.

Execution Environment

The Execution Environment manages sessions where you execute TIBCO Object Service Broker rules and access TIBCO Object Service Broker tables. The Execution Environment has two parts: the TIBCO Object Service Broker monitor (osMon) process and the osee program.

The osMon process creates, manages, and terminates sessions. It establishes links with entities outside the Execution Environment. You must run osMon before starting a TIBCO Object Service Broker session and should terminate it only after all sessions are done. When you request the start of a TIBCO Object Service Broker session, osMon creates an osee as necessary for your use. The osee program delegates actions to a session it has created and supervises the session while it exists. Execution Environments are either single- or multiple-session.

More than one Execution Environment can reside on one machine, and multiple Execution Environments can interact with the same Data Objext Broker. Although one session can interact with only one Data Object Broker directly, it can interact with other Data Object Brokers through distributed data access between Data Object Brokers.

Data Object Broker

The Data Object Broker handles the co-ordination and management of transactional table data. It acts as the transactional commit coordinator, and in this capacity manages the integrity of transactional data. It can also route data access traffic to another Data Object Broker or to an external database server. The logical view of the data that it manages is kept in the MetaStor.

Physical Data Store

The physical data store, known as the Pagestore, is where the actual data is stored on a physical device in a device dependent format. TIBCO Object Service Broker makes use of this device dependent format to store its logical, relational table view of data.

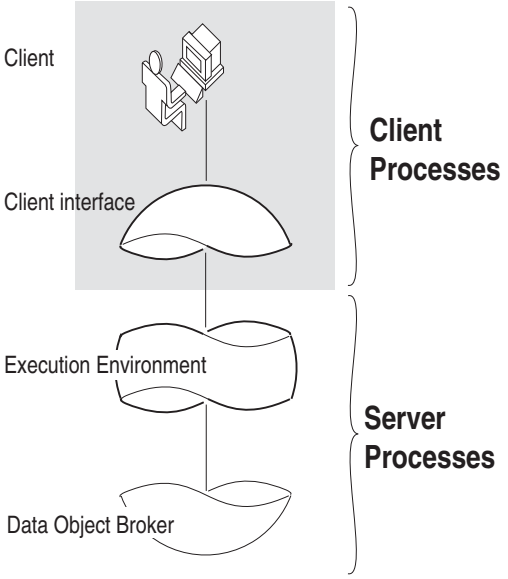
External Database Servers

External database servers allow TIBCO Object Service Broker to access other types of data on external databases. For detailed information about external database servers, refer to the *TIBCO Service Gateway* manual that accompanies each external database server.

See Also *TIBCO Object Service Broker for Open Systems Installing and Operating* for information about configuration setup.

TIBCO Object Service Broker in the Client Server Model

The client/server model reflects the relationship between a service requester (client) and a service provider (server). The basic division of client processes and server processes as applied to the TIBCO Object Service Broker architecture is shown in the following diagram:



The two main components of the client/server relationship within TIBCO Object Service Broker are the client interface layer and the Execution Environment. The client interface layer includes TIBCO Object Service Broker clients and other interfaces for external environments.

See Also *TIBCO Object Service Broker for Open Systems Installing and Operating* for information about installing the TIBCO Object Service Broker clients.

TIBCO Object Service Broker Interfaces for External Environments

TIBCO Object Service Broker provides a way for other programs and applications to make use of TIBCO Object Service Broker services. These external clients do not have to exist in the same operating environment, and must be able to take advantage of this interface:

ODBC	A Microsoft standard application programming interface to TIBCO Object Service Broker, described in Chapter 6, Using TIBCO Object Service Broker Adapter for JDBC-ODBC, on page 75, for accessing an Execution Environment from a remote system
JDBC	A Java standard application programming interface to TIBCO Object Service Broker, described in Chapter 6, Using TIBCO Object Service Broker Adapter for JDBC-ODBC, on page 75, for accessing an Execution Environment from a remote system
SDK (C/C++)	An application programming interface to TIBCO Object Service Broker, described in Chapter 7, Using TIBCO Object Service Broker SDK (C/C++), on page 109, for accessing an Execution Environment from a remote system
SDK (Java)	An application programming interface to TIBCO Object Service Broker, described in Chapter 8, Using TIBCO Object Service Broker SDK (Java), on page 143, for accessing an Execution Environment from a remote system in a Java environment
TIBCO Object Service Broker UI	An Eclipse-based UI that provides a graphical environment for TIBCO Object Service Broker development and to access TIBCO Object Service Broker data. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.
Telnet 3270	With Telnet 3270, you can use external 3270 emulation programs to get a text-only interface to access TIBCO Object Service Broker data. For further information, refer to *TIBCO Object Service Broker Getting Started*.

Accessing TIBCO Object Service Broker from an External Environment

What is an External Environment?

All programs and applications run on a specific operating system platform (an environment). Although written in different languages with different areas of focus, they are sustained by and commonly share the resources of the environment where they run.

TIBCO Object Service Broker is Open to Its External Environment

In a similar way, TIBCO Object Service Broker makes its data and resources available to those applications that can make use of Object Integration Gateway, the SDK (C/C++), SDK (Java), Telnet 3270, ODBC, or JDBC interfaces.

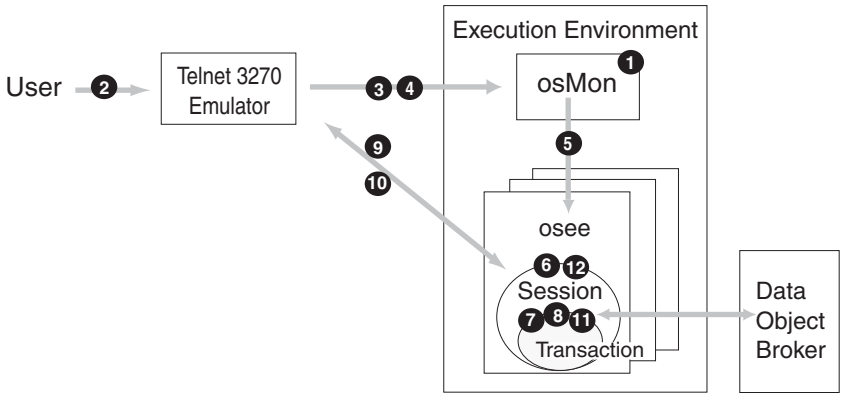
General Steps

For Telnet 3270

1. The system administrator starts the TIBCO Object Service Broker monitor process (osMon).
2. The user starts their Telnet 3270 Emulator program, supplying the name of the host where the active osMon resides and the number of the port where this osMon is listening.
3. On the login screen that appears, they enter their user ID and password, and any session parameters that they want to have override those in session.prm.
4. The Telnet 3270 Emulator program tells osMon, via Telnet 3270, that it wants to start a session and run the first rule. The client passes the session parameters to the osMon.
5. If no instance of an osee with the requested name is available, osMon starts one.
6. The instance of osee from the previous step starts a session with the parameters passed from the client during step #4..
7. The session starts a transaction and runs the first rule.
8. The rule populates screen tables and issues a DISPLAY statement.
9. The screen is delivered to the client and appears on the client console.

10. The user modifies text on the console and presses a PF key or Enter. Modified screen tables, along with the name of the PF key that the user pressed, are made available to the rule. The rule continues its execution after the DISPLAY statement.
11. When the rule is done, the session stops the transaction created on step 6.
12. osee stops the session.

Depending on the algorithm in the rule, step 7 to step 10 can be repeated any number of times.

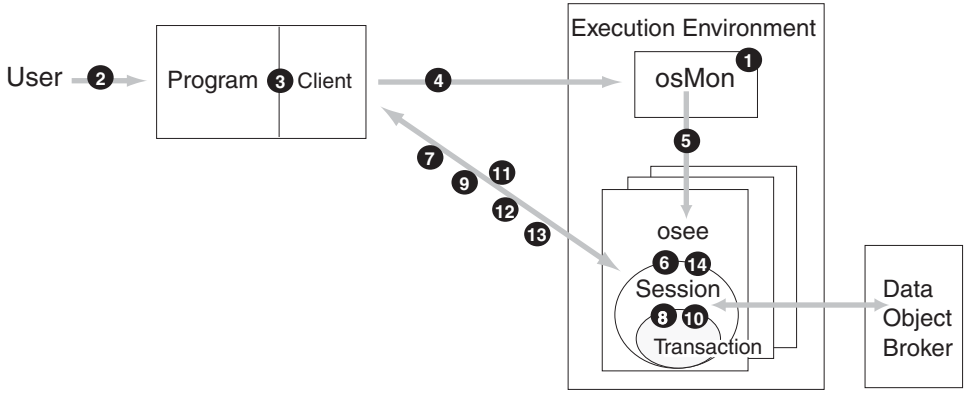


For SDK (C/C++)

1. The system administrator starts the TIBCO Object Service Broker monitor process (osMon).
2. The user starts the application program.
3. The program requests a session startup by issuing a STARTSS SDK (C/C++) request, supplying session parameters that include the name of the Execution Environment to host the session.
4. The SDK (C/C++) client tells osMon that it wants to start a session with the supplied session parameters.
5. If no instance of an osee with the requested name is available, osMon starts one.
6. The instance of osee from the previous step starts a session with parameters passed by the SDK (C/C++) client during step #4..
7. The program requests a new transaction by issuing a STARTTR.
8. The session starts a transaction.

9. The program asks for a rule to be called by issuing a CALLRULE supplying a rule name, parameters, and optionally some commarea data.
10. The session calls the rule.
11. On completion of the rule, all the return information, possibly including output commarea data, is delivered to the client.
12. The program asks the session to stop the current transaction, committing or rolling back any changes made by the rule, by issuing a STOPTR call.
13. The program asks the session to terminate, by issuing a STOPSS call.
14. The osee program stops the session.

Steps #3. to #13. can be repeated any number of times according to the program algorithm. The order of these steps is not essential as long as it complies with basic SDK (C/C++) sequencing rules. Also, a program can work with any number of sessions at any time.

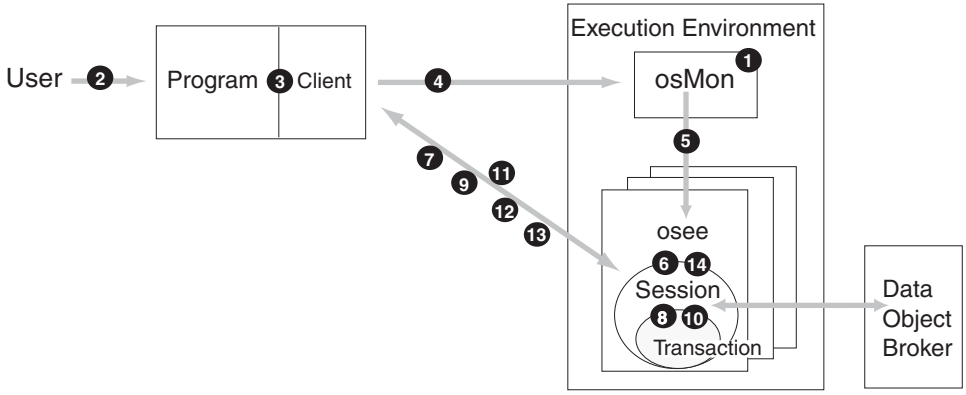


For SDK (Java)

1. The system administrator starts the TIBCO Object Service Broker monitor process (osMon).
2. The user starts the application program.
3. The program requests a session startup by creating a Session object using the second form of the Session constructor or by calling a start method on a previously created Session object, supplying session parameters that include the name of the Execution Environment to host the session.
4. The SDK (Java) client tells osMon that it wants to start a session with the supplied session parameters.

5. If no instance of an osee with the requested name is available, osMon starts one.
6. The instance of osee from the previous step starts a session with parameters passed by the SDK (Java) client during step #4..
7. The program requests a new transaction by calling startTrans.
8. The session starts a transaction.
9. The program asks for a rule to be called by calling a call method supplying a rule name, parameters, and optionally some commarea data.
10. The session calls the rule.
11. On completion of the rule, all the return information, possibly including output commarea data, is delivered to the client.
12. The program asks the session to stop the current transaction, committing or rolling back any changes made by the rule, by issuing a stopTrans call.
13. The program asks the session to terminate, by issuing a stop call.
14. The osee program stops the session.

Steps #3. to #13. can be repeated any number of times according to the program algorithm. The order of these steps is not essential as long as it complies with basic SDK (Java) sequencing rules. Also, a program can work with any number of sessions at any time.



For osBatch

Because osBatch embeds the functionality of osee, the osBatch execution flow is different from all other clients.

1. The user starts osBatch, supplying a parameter string that includes Execution Environment parameters, including the name of the Data Object Broker to connect to, and session parameters, including the first rule name.
2. osBatch initializes using the Execution Environment parameters supplied by the user.
3. osBatch starts a session using the session parameters supplied by the user.
4. The session starts a transaction and runs the first rule.
5. After the rule is done, the session stops the transaction created in step #4.
6. osBatch stops the session.
7. osBatch terminates.

Chapter 2 Setting Session Parameters

This chapter describes how to set session parameters, how to start a session, and what to do when a session ends abnormally.

Topics

Session Parameter Setting

What is a Session?

A session is a unit of resources that permits a user to run TIBCO Object Service Broker transactions. To create a session, users identify themselves through a TIBCO Object Service Broker user ID and specify the runtime attributes for the session. This identification and specification take place through the use of session parameters. When a session ends, all resources held by the session on behalf of a user are released.

How to Set Session Parameters

Use the following table to determine how to specify the parameter values of your session and where to get additional information about setting the values. The parameters that you can set are described in detail in *TIBCO Object Service Broker Parameters*.

How to Set Parameters …	Refer to …
Using TIBCO Object Service Broker SDK (C/C++).	Chapter 7, Using TIBCO Object Service Broker SDK (C/C++), on page 109.
Using TIBCO Object Service Broker SDK (Java).	Chapter 8, Using TIBCO Object Service Broker SDK (Java), on page 143.
Using the User Profile option of the developer workbench, or the Security Manager option of the administrator workbench.	*TIBCO Object Service Broker Managing Security*
Using parameter files.	*TIBCO Object Service Broker Parameters*
Using Object Integration Gateway.	*TIBCO Object Service Broker Object Integration Gateway*

Starting Sessions

How Do You Start Sessions?

A session is started by a client. The client can be one that is supplied by TIBCO Object Service Broker or an external Telnet 3270 Emulator program connected to the TIBCO Object Service Broker 3270 Access Adapter. The following table lists these clients and where you can get additional information about starting a session:

Client	Refer to…
3270 Access Adapter	*TIBCO Object Service Broker Getting Started*
TIBCO Object Service Broker UI	*TIBCO Object Service Broker Getting Started*
Object Integration Gateway	*TIBCO Object Service Broker Object Integration Gateway*
TIBCO Object Service Broker Adapter for JDBC-ODBC	Chapter 6, Using TIBCO Object Service Broker Adapter for JDBC-ODBC, on page 75.
osBatch	*TIBCO Object Service Broker for Open Systems Utilities*
ostty	*TIBCO Object Service Broker Getting Started*
SDK (C/C++)	Chapter 7, Using TIBCO Object Service Broker SDK (C/C++), on page 109.
SDK (Java)	Chapter 8, Using TIBCO Object Service Broker SDK (Java), on page 143.

When a Session Ends Abnormally

List of Execution Environment and Session Exit Codes

To help in diagnosing a failure, here is a list of codes that you can encounter, with some action that you can take to fix the problem:

Code	Description	Action
0	The session exited successfully.	No action is required.
128	The session stopped as a result of an operator request or of a Data Object Broker request.	No action is required.
129	The Execution Environment stopped as a result of an operator request.	No action is required.
130	The Execution Environment closed as a result of an operator request.	No action is required.
143	The first rule of the session ended as a result of an uncaught exception.	Check the session logs. Review your application. Place the appropriate exception handler at the end of your first rule to catch the exception.
154	No session startup rule is supplied.	Add the startup rule parameter to your session parameters.
155	You attempted to start a session with no user ID.	Add the USERID parameter to your startup parameters.
156	You specified an invalid password in the startup parameters.	Restart your session with the password that corresponds to the USERID parameter.
157	You attempted to start a session during a restricted time.	Your user account is set up so that it can log in to the system only during certain hours of the day. You should either get your account updated, or wait until you are permitted to log in.

Code	Description	Action
158	The TIBCO Object Service Broker user ID does not match the operating system user ID.	System security is set to require the TIBCO Object Service Broker user ID to match the user's operating system user ID. Either change the security setting or log in with matching user IDs.
159	Your user account is suspended.	Ask your system administrator to lift the suspension from your account connection.
167	The Execution Environment to Data Object Broker internal connection limit is exceeded.	Contact TIBCO Support.
168	The peer server failed to log in to the Data Object Broker.	Check the following: • Whether the Data Object Broker is started • Whether the Data Object Broker host is accessible via your network • That the DOB Execution Environment parameter is correct • That the SERVERS Execution Environment parameter is correct • That the PEERS Data Object Broker parameter is correct in the crparm file • Check the Data Object Broker node definition in the TIBCO Object Service Broker directory file (huron.dir) Finally, check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
169	The transaction ended because it required more storage.	Increase the value of the TRANMEMMAX Execution Environment parameter.
171	Initialization of the NLS system failed.	Check the contents of the @NLS1 table. If necessary, also check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
173	The session ended because it required more storage.	Increase the value of the SESSIONMEMMAX Execution Environment parameter.

Code	Description	Action
174	TIBCO Object Service Broker cannot locate the path specified by the OS_ROOT environment variable.	Correct the value of OS_ROOT environment variable.
175	The audit log update failed.	Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any possible clue.
176	The prompt for user ID and password failed.	No action is required.
191	The TIBCO Object Service Broker session failed to log in to the Data Object Broker.	Check the following: • Whether the Data Object Broker is started • Whether the Data Object Broker host is accessible via your network • That the DOB Execution Environment parameter is correct • That the MAXUSERS Data Object Broker parameter is correct in the crparm file • Check the Data Object Broker node definition in the TIBCO Object Service Broker directory file (huron.dir) Finally, check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
234	TIBCO Object Service Broker failed to establish a connection to the client process.	Check the operating system diagnostics for a possible socket problem. Check your network connectivity.
235	The connection to the client process broke.	If the client process was not explicitly killed, check your network connectivity. Otherwise, no action is required

Code	Description	Action
236	The TIBCO Object Service Broker session failed to open a connection to the Data Object Broker.	Check the validity of the Data Object Broker node definition in the TIBCO Object Service Broker directory file (huron.dir). Check whether the Data Object Broker host is accessible via network. Finally, check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
237	The connection to the Data Object Broker process broke.	Check whether the Data Object Broker host is still accessible via the network. Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
238	TIBCO Object Service Broker failed to establish a peer server connection to the Data Object Broker process.	Check the SERVERS Execution Environment parameter. Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
239	The peer server connection to the Data Object Broker broke.	Check whether the Data Object Broker host is still accessible via the network. Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any other possible clue.
253	The Execution Environment terminated due to a failure on its listening named pipe.	Check the operating system diagnostics for possible interprocess communication problems. For assistance, contact TIBCO Support.
254	Execution Environment or session initialization failed.	Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any possible clue. For assistance, contact TIBCO Support.
255	The session failed with an unknown error.	Check the log files (Data Object Broker, Execution Environment, osMon, and session) for any possible clue. For assistance, contact TIBCO Support.

Chapter 3 Accessing External Routines

This chapter describes how to access external routines from TIBCO Object Service Broker.

Topics

Overview

Functional Overview

How Does TIBCO Object Service Broker Process an External Routine?

From TIBCO Object Service Broker you can pass control to and receive control back from a routine outside TIBCO Object Service Broker operating boundaries. When an external routine is called, the following takes place:

1. The calling rule goes into a wait state.
2. TIBCO Object Service Broker calls the external routine and waits for control to return to it.

The link to the external routine remains until the session ends.

What Languages are Supported?

TIBCO Object Service Broker for Open Systems supports external routines in C and Java.

C

For routines in C, the following conditions must be met:

- The routine must be written in a language whose implementation is compatible with the required C prototype.
- The routine must be compiled as 32-bit code.

The C compiler must be able to produce:

- A relocatable, executable file with some symbols left to be resolved at runtime
- A dynamically linked library entry

Java

Java routines must be static Java methods.

Even though TIBCO Object Service Broker allows only static methods as external routines, non-static methods of a particular target class can be invoked indirectly, as shown here:

1. Define a (mapping) class that manages a mapping of handles (for example, integer values) to instances of a target class.

The mapping should be maintained in a static data structure within that class.

2. Have the mapping class implement a static method (a class factory) that creates instances of the target class and returns a handle to the instance while adding, to the internal data structure, the mapping from the handle to the instance.

   This class factory method can be called by a TIBCO Object Service Broker application to create instances of the target class.

3. For each instance method of the target class that you want to invoke from the TIBCO Object Service Broker application, have the mapping class implement an intermediate static method. This method takes a target class instance handle and arguments corresponding to the arguments of the instance method to be invoked.

4. Call this intermediate static method from your TIBCO Object Service Broker application.

   The intermediate method should resolve the handle into a class instance reference and call the instance method, passing the required arguments to it.

The mapping class methods should also check handles for validity and possibly implement further (static) methods. These methods are then available for a TIBCO Object Service Broker application to manage the life cycle of the target class instances.

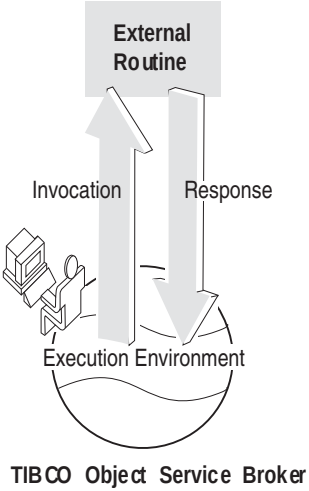
Syntax for Calling the Routine

The external routine can be invoked from within a rule either as a function or explicitly with the CALL statement as in the following examples:

Explicit Invocation	Invocation As a Function
`CALL USERPROC(string);`	`totalcost = USERFUNC (1.99, .07);`
`CALL EXT_ROUTINE_A;`	`Y = 2*EXTERNAL_R6(arg1, arg2, arg3);`

Process Flow

The following shows the process flow between TIBCO Object Service Broker and an external routine.



See Also *TIBCO Object Service Broker Programming in Rules* for information about coding rules and the rules language

External Routines in C

Steps Required to Use an External C Routine

Syntax Mapping

The osextusr.h required header file, available in the install_path\extrc\include folder, is used to map TIBCO Object Service Broker syntax to C syntax. It contains a list of the supplied functions used to manipulate the arguments passed to TIBCO Object Service Broker. It must be in your include library during compilation.

How are Exceptions Handled?

There are no exceptions trapped during external routine invocation and execution, such as, if the external routine is not found or it fails. A message is written to the message log and a traceback is generated if a failure occurs.

Procedural Overview

The major tasks in preparing and executing an external C routine are:

1. Coding, compiling, and linking your program, page 23
2. Identifying the external routine to TIBCO Object Service Broker, page 27

When these tasks are completed, a TIBCO Object Service Broker rule can invoke the external routine.

If your Execution Environment is multi-threaded (that is, the MAXSESSION Execution Environment parameter is greater than 1), your external routines must be reentrant and thread-safe.

Task A Coding, compiling, and linking your program

Coding the External Routine

The external routine implementation must comply with the following C prototype:

```
void user_routine (HRN_EXT_PARAM param);
```

The same prototype is used whether your routine is a function or a straight procedure.

Key Elements The key elements in the prototype are:

user_routine	This name must be exported exactly as it appears in the **LOADNAME** field of the ROUTINES table. Refer to Identifying the external routine to TIBCO Object Service Broker, page 27 on page 23 for more information.
param	Used to pass arguments for the external routine.

Sample Routines for Compiling and Linking under Windows The following contains a set of sample external routines for Windows. As noted in the comments, they implement a function rule and a procedure rule:

```
/*************************************************************/
/*                 Sample External Routines                  */
/*                                                           */
/*************************************************************/

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

/* Include all the supplied access routines for manipulating
   the arguments passed from TIBCO Object Service Broker.   */

#include "osextusr.h"

/*************************************************************
   This function accepts two arguments: a cost and a tax rate.
   It returns the cost with the tax applied. If an access
   routine fails (indicated by the return of a value of -1)
   the external routine returns immediately.
*************************************************************/

void userFunc (HRN_EXT_PARAM param)
{
   double cost;
   double tax_rate;
   double taxed_cost;

   /* Get first argument. */
   if (hrnGetDoubleArg (param, 1, &cost))
     {
       return;
     }

   /* Get second argument. */
   if (hrnGetDoubleArg (param, 2, &tax_rate))
      {
         return;
```

```
      }

   /* Calculate total cost with tax. */
   taxed_cost = cost + cost * tax_rate;

   /* Set returned value. Returned value is placed in
      parameter in position 0.*/
   if (hrnSetDoubleArg (param, 0, taxed_cost))
      {
         return;
      }
}

void userProc (HRN_EXT_PARAM param)

/*************************************************************
   This function accepts one arguments: a string. It modifies
   the string by concatenating "HELLO WORLD". If an access
   routine fails (indicated by the return of a value of -1) the
   external routine returns immediately.
*************************************************************/
{
   char      *pszstring;
   char      *concat_string = "HELLO WORLD";
   int      concat_string_length = 11;
   int      string_length;
   /* Get the length of the string argument.
   If a negative string length is returned,
   the access function failed - return immediately. */
   string_length = hrnGetCharArgLen (param, 1);

   if (string_length <= 0)
      {
         return;
      }

   /* Allocate and initialize memory for the length of the
   string plus the concatenation string length plus the
   trailing '\0' character. Return if unable to allocate the
   memory.                                                  */

   pszstring = malloc (
      string_length + concat_string_length + 1);
   if (!pszstring)
      {
         return;
      }
   memset (pszstring, 0,
      string_length + concat_string_length + 1);

   /* Get string argument. Concatenate "HELLO WORLD" to
   the string.                                              */

   if (hrnGetCharArg (param, 1, pszstring,
      string_length + 1))
      {
         return;
      }

   strcat (pszstring, concat_string);
```

```
/* Replace the argument with the modified string.*/
   if (hrnSetCharArg (param, 1, pszstring))
      {
         return;
      }
}
```

Sample Import/Export Definitions File The sample code for the Import/Export definitions (*test.def*) is:

```
LIBRARY Test
EXPORTS
      USERFUNC = userFunc
      USERPROC = userProc
```

In this example, the mixed-case symbol names `userFunc` and `userProc` are exported uppercase (`USERFUNC` and `USERPROC`), as required by **LOADNAME** field of the ROUTINES table. Refer to Identifying the external routine to TIBCO Object Service Broker, page 27 on page 23 for more information.

Sample Routines for Compiling and Linking under Solaris The following contains a set of sample external routines for Solaris:

```
#include "osextusr.h"

#if defined(__cplusplus)
extern "C" {
#endif

void EXTRTEST (HRN_EXT_PARAM param)
{
   int op1;
   int op2;
   int sum;

   if (hrnGetIntArg (param, 1, &op1))
      return;

   if (hrnGetIntArg(param, 2, &op2))
      return;
   sum = op1 + op2;

   if (hrnSetIntArg(param, 0, sum))
      return;
}

#if defined(__cplusplus)
} // extern "C"
#endif
```

Compiling and Linking

The routine must be compiled as 32-bit code by a compiler that can produce:

- A relocatable, executable file with some of its symbols unresolved until runtime
- An entry for a dynamically linked library

External Routine for the Windows Platform The external routine DLL can be built with any compiler able to produce 32-bit code compatible with the Microsoft Visual C++ runtime environment. The DLL must be linked with the osextusr.lib import library.

Compiling and Linking External Routines on Solaris On Solaris, external routines are compiled as shared libraries. Use the following commands to create a C++ version of the shared libraries.

In this example, *install_path* refers to the TIBCO Object Service Broker installation folder:

```
g++ -c -o extrtest.o -fPIC -D_POSIX_C_SOURCE=199506L\
   -U_XOPEN_SOURCE -D_XOPEN_SOURCE -D__EXTENSIONS__ -D_REENTRANT\
   -I<install_path>/extrc/include extrtest.c
g++ -o libextrtest.so -shared extrtest.o
```

Dynamic Linking

So that external routines can call functions external to TIBCO Object Service Broker, the libraries containing these functions must reside in the search path for osMon. In Windows, this is in the PATH environment variable. In Solaris, this is in the LD_LIBRARY_PATH.

Task B Identifying the external routine to TIBCO Object Service Broker

Specify the Table Entries

The external routine and its load module name are identified to TIBCO Object Service Broker through an entry in the ROUTINES table. If the routine has arguments, these are specified in a table instance of the ARGUMENTS table. Ensure that you have adequate security to insert data into these tables before editing them.

Use the Table Editor to add information to these tables. To use this option, enter the required table name beside the ED edit table option on the workbench and press Enter to display the table.

Add an Entry in the ROUTINES Table

The following illustration shows an extract of the ROUTINES table. You must scroll right using PF11 to see additional fields.

```
  EDITING TABLE   :   ROUTINES
COMMAND ==>
                                                                SCROLL: P
        NAME            LANGUAGE      FUNCTION TYPE SYNTAX LENGTH DECIMAL
_ ---------------- ----------------      -      -      -     ------ ------
_ ABEND                                  N                        0      0
_ ADMCHART                               N                        0      0
_ ASMS_ASTRANSLATE                       N                        0      0
_ ASREAD                                 N                        0      0
_ BINARY_TO_LOGIC                        Y      L      C          1      0
_ BTOPACKD                               Y             C          1      0
_ CATROW                                 N                        0      0
_ CCOB11                                 N                        0      0
_ CCOB12A                                N                        0      0
_ CCOB12B                                N                        0      0
_ CDIR50O                                N                        0      0
_ USERFUNC                               Y      Q      P          5      2
_ USERPROC                               N                        0      0


PFKEYS: 4=INSERT 16=DELETE 5=FIND NEXT 6=CHG NEXT 18=EXCLUDE 3=SAVE 12=CANCEL
```

Type the appropriate information about your external routine in fields of the ROUTINES table, as described:

NAME	The name used to invoke the external routine. Syntax: C, Length: 16
LANGUAGE	The language in which the external routine is written. This field is used for descriptive purposes only and is not required. Syntax: C, Length: 16
FUNCTION	Whether the external routine returns a value (Y or N). Syntax: C, Length: 1
TYPE	If the external routine is a function, specify the TIBCO Object Service Broker semantic type of the value returned. Syntax: C, Length: 1

SYNTAX	If the external routine is a function, specify the TIBCO Object Service Broker syntax of the value returned. Syntax: C, Length: 1
LENGTH	If the external routine is a function, specify the length of the value returned. Syntax: B, Length: 2
DECIMAL	If the external routine is a function that returns a value with digits to the right of the decimal, specify the number of digits. Syntax: B, Length: 2
LOADNAME	The name of the entry point from where the DLL should be started. Syntax: C, Length: 8 This name *must* be exported uppercase in the exports definition file. The symbol to be loaded can be lowercase or mixed case in the external routine, but it must be exported uppercase to the **LOADNAME** field.
SCOPE	The point when the external library is unloaded. At present the only valid value is SESSION (that is, the external library must be unloaded when the session ends).
NODENAME	NODENAME can be either left blank or set to @SESSION, meaning that the C routine executes in the same process as the Execution Environment. Syntax: V, Length: 255
LIBNAME	Specify the full path name of the C routine. It must be a dynamic link library for Windows or a shared object library for Solaris. Syntax: V, Length: 255

Sample Entry

The single occurrence from the ROUTINES table for the routine XYZ follows. In this example USERFUNC, whose source code you can find in Coding the External Routine on page 23, is called as a function.

```
--- SINGLE OCCURRENCE EDITOR ---
 EDITING TABLE    : ROUTINES
 TABLE TYPE       : TDS
 COMMAND ==>
 ------------------------------------------------------------------------------

 NAME             : USERFUNC
 LANGUAGE         : C
 FUNCTION         : Y
 TYPE             : Q
 SYNTAX           : P
 LENGTH           :       5
 DECIMAL          :       2
 LOADNAME           USERFUNC
 SCOPE            : Session
 NODENAME         : @SESSION
                  :
                  :
                  :
                  :
LIBNAME          : D:\Test\dir\test.dll
                  :
 PFKEYS: 1=HELP 2=DOCUMENTATION 3=SAVE 12=CANCEL 13=PRINT 22=DELETE
```

Add an Entry in the ARGUMENTS Table

If your external routine has arguments, to identify the arguments to TIBCO Object Service Broker add a table instance to the ARGUMENTS table. The parameter value for the table instance of the ARGUMENTS table must be the name of the external routine (that is, the value in the **NAME** field of the ROUTINES table). You can have a maximum of 16 arguments.

The instance of the ARGUMENTS table for the sample routine XYZ follows:

```
 EDITING TABLE    :    ARGUMENTS(USERFUNC)
COMMAND ==>
                                                                  SCROLL: P
   NUMBER        NAME         INOUT TYPE SYNTAX LENGTH DECIMAL
_ ------ ----------------      -     -      -    ------ ------
_       1 COST                  N     Q      P         5      2
_       2 TAX_RATE              N     Q      P         2      2

PFKEYS: 4=INSERT 16=DELETE 5=FIND NEXT 6=CHG NEXT 18=EXCLUDE 3=SAVE 12=CANCEL
At TOP
```

Type the appropriate information in the fields of the ARGUMENTS table:

NUMBER	The position of the argument in the argument list. The positions must be sequential and start at 1. Syntax: B, Length: 2
NAME	The argument name. Syntax: C, Length: 16
INOUT	Whether the value of the argument can be changed by the external routine. If it can be changed, the value passed to the routine must be a local variable or the field of a table and the field must have the same data definition as the argument. Valid entries are Y or N. Syntax: C, Length: 1
TYPE	The argument semantic data type. Syntax: C, Length: 1
SYNTAX	The argument syntax. Syntax: C, Length: 1

LENGTH	The argument length. Syntax: B, Length: 2
DECIMAL	The number of digits to the right of the decimal, if any. Syntax: B, Length: 2

See Also

TIBCO Object Service Broker Managing Data for information about how to use the Table Editor.

TIBCO Object Service Broker Programming in Rules about valid TIBCO Object Service Broker syntax and semantic data types.

External Routines in Java

Steps Required to Use an External Java Routine

TIBCO Object Service Broker String Conversions

Passing a TIBCO Object Service Broker String to a Java External Routine

A string is passed to a Java external routine as an argument in one of two ways.

If the corresponding Java argument is a one-dimensional array of type byte, the following takes place:

1. A Java byte array is created.
2. The bytes in the TIBCO Object Service Broker string argument are used to populate the array.
3. The array is passed to the Java routine.
4. The Java routine is invoked.
5. The array is released for garbage collection when the routine returns.

If the corresponding Java argument is of the predefined class String, the following takes place:

1. A String class object is created and initialized using a constructor that accepts a byte array and a code page indicator. The constructor interprets the TIBCO Object Service Broker string passed as a byte array as a single-byte character string in the code page indicated by the String argument and causes the value of the String object to become the result of converting the single-byte character string to UNICODE.
2. The String class object is passed to the Java routine.
3. The routine is invoked.
4. The String class object and other intermediate objects are released for garbage collection when the routine returns.

Returning a TIBCO Object Service Broker String from a Java External Routine

A TIBCO Object Service Broker string is returned from a Java external routine as a return value in one of two ways:

- If the corresponding Java return value is a one-dimensional array of type byte, the bytes of that array are extracted and used to form the characters of the string returned to the calling TIBCO Object Service Broker rule.
- If the corresponding Java return value is of class String, a single-byte character string in the Execution Environment code page is extracted from the String using a method that returns a byte array containing the character string converted from UNICODE using the code page designation passed to it.

How are Exceptions Handled?

If a Java external routine raises a Java exception, the TIBCO Object Service Broker JAVAFAIL exception is raised upon return from the external routine. This exception can be caught in rules code and the error message and logs generated by the Java exception can be checked.

Procedural Overview

The major tasks in preparing and executing an external Java routine are:

- Prepare the external program and environment, page 34
- Identify the external routine to TIBCO Object Service Broker, page 36

When these tasks are completed, a TIBCO Object Service Broker rule can invoke the external routine.

If your Execution Environment is multi-threaded (that is, the MAXSESSION Execution Environment parameter is greater than 1), your external routines must be reentrant and thread-safe.

Task A Prepare the external program and environment

Compiling Your Java Class

After writing your Java class, compile it using a Java version 1.6 or later compiler.

Java Virtual Machine

A Java Virtual Machine (JVM) must be invoked and attached to an Execution Environment before a rule can execute a Java external routine. Each Execution Environment can specify which JVM to attach, allowing different Java routines to use different versions of JVM.

The JVM used must support code page Cp037.

You specify the JVM and the options to pass to it in the @JAVAOPTIONS table, which is parameterized by EENAME:

```
Table: @JAVAOPTIONS     Type: TDS   Unit: JAVA              IDgen: N
  Parameter Name  Typ Syn Len Dec Class            '      Event Rule   Typ Acc
 ----------------  -   -  --- --   -              '   ---------------- -   -
 EENAME             S   C   32  0    D             ' _
 LOCATION           I   C   16  0    L             ' _
    Field Name    Typ Syn Len Dec Key Ord Rqd    Default          Reference
 ---------------- -   -  ---- --  -   -   - ---------------- ----------------
 SEQ#               I   B     2  0  P
 TYPE               S   C     1  0                              @JAVAOPTIONTYPE
 VALUE              S   V    64  0
```

When the first Java external routine is called after Execution Environment start-up, by default the system searches for the EENAME instance that matches the Execution Environment in use. If this instance does not exist, the instance with EENAME equal to "*DEFAULT*" is used instead.

The SEQ# key field of @JAVAOPTIONS indicates the sequence in which the occurrences are processed. The values of the SEQ# field do not have to be contiguous.

There are 3 types of occurrences in the table:

- JVM location occurrences (indicated by a TYPE field equal to L).

  There must be one and only one location occurrence in each instance of the table. The VALUE field in this occurrence contains the path to the JVM to be started. If the path is longer than this 64-character field, one or more continuation occurrences follow, containing the remainder of the path in their VALUE fields.

- JVM command line parameter occurrences (indicated by TYPE field equal to P).

  There can be zero or more parameter occurrences in each instance of the table. A parameter occurrence contains a command line parameter that must be passed to the JVM. Such parameters can be used to specify storage allocations or debugging options to the JVM. The options are specific to the implementation of the JVM and their names, form and meaning are specified in the documentation of the JVM in question.

- Continuation occurrences (indicated by TYPE field equal to C).

Here is an example of an instance of @JAVAOPTIONS on Windows:

```
SEQ#          TYPE                                VALUE
------         -   ------------------------------------------------------------
10             L   E:\jdk1.6.0_30\jre\
15             C   bin\server\jvm.dll
20             P   -Djava.class.path=e:\testjava
30             P   -verbose:class,gc
```

When an Execution Environment calls a Java external routine by using this instance to determine the JVM to start, it attempts to load the following DLL:

```
E:\jdk1.6.0_30\jre\bin\server\jvm.dll
```

and pass it these parameters:

```
-Djava.class.path=e:\testjava -verbose:class,gc,jni
```

which, in the case of the Sun JVM for Windows cause it to use a CLASSPATH variable value of e:\testjava and runtime options that indicate tracing output of class loading and garbage collection is required.

The JVM remains attached to the Execution Environment until the Execution Environment terminates.

Task B Identify the external routine to TIBCO Object Service Broker

Specify the Table Entries

The external routine and its load module name are identified to TIBCO Object Service Broker through an entry in the ROUTINES table. If the routine has arguments, these are specified in a table instance of the ARGUMENTS table. Ensure that you have adequate security to insert data into these tables before editing them.

Use the Table Editor to add information to these tables. To use this option, enter the required table name beside the ED edit table option on the workbench and press Enter to display the table.

Add an Entry in the ROUTINES Table

The following illustration shows an extract of the ROUTINES table. You must scroll right using PF11 to see additional fields.

```
  EDITING TABLE   :   ROUTINES
COMMAND ==>
                                                                  SCROLL: P
        NAME            LANGUAGE     FUNCTION TYPE SYNTAX LENGTH DECIMAL
_ ---------------- ----------------    -       -      -     ------ ------
_ ABEND                                N                        0      0
_ ADMCHART                             N                        0      0
_ ASMS_ASTRANSLATE                     N                        0      0
_ ASREAD                               N                        0      0
_ BINARY_TO_LOGIC                      Y       L      C         1      0
_ BTOPACKD                             Y              C         1      0
_ CATROW                               N                        0      0
_ CCOB11                               N                        0      0
_ CCOB12A                              N                        0      0
_ CCOB12B                              N                        0      0
_ CDIR50O                              N                        0      0
_ USERFUNC                             Y       Q      P         5      2
_ USERPROC                             N                        0      0


PFKEYS: 4=INSERT 16=DELETE 5=FIND NEXT 6=CHG NEXT 18=EXCLUDE 3=SAVE 12=CANCEL
```

Type the appropriate information about your external routine in fields of the ROUTINES table, as described:

NAME	The name used to invoke the external routine. Syntax: C, Length: 16
LANGUAGE	The language in which the external routine is written, in this case JAVA. Syntax: C, Length: 16
FUNCTION	Whether the external routine returns a value (Y or N). Syntax: C, Length: 1
TYPE	If the external routine is a function, specify the TIBCO Object Service Broker semantic type of the value returned. Syntax: C, Length: 1

SYNTAX	If the external routine is a function, specify the TIBCO Object Service Broker syntax of the value returned. Syntax: C, Length: 1
LENGTH	If the external routine is a function, specify the length of the value returned. Syntax: B, Length: 2
DECIMAL	If the external routine is a function, specify the value zero, because Java does not support packed decimal values as a basic type. Syntax: B, Length: 2
LOADNAME	Leave this field empty. Syntax: C, Length: 8
SCOPE	The point when the external library is unloaded. At present the only valid value is SESSION (that is, the external library must be unloaded when the session ends).
NODENAME	Leave this field empty. Syntax: V, Length: 255
LIBNAME	Specify the name of the Java class containing the routine, the name of the routine (that is, the method within the class), and the Java type of the arguments and the return value (if any). Syntax: V, Length: 255

This entry follows the format:

```
{ <package name> "/" } <class name> "." <method name> "(" <argument
types> ")" <return type>
```

where <*package name*>, <*class name*> and <*method name*> are the ASCII equivalents of the UNICODE names of the package, class and method, <*argument types*> is a (possibly empty) character sequence encoding the Java types of the arguments, and <*return type*> is an encoding of the return value. The string beginning with left parenthesis and spanning the rest of the specification is called the signature of the method.

Java Types

A basic Java type is encoded with a single character as follows:

Character	Java Type	Interpretation
B	Byte	Signed byte.
C	Char	Unicode character.
D	Double	Double-precision floating-point value.
F	Float	Single-precision floating-point value.
I	Int	Integer.
J	Long	Long integer.
S	Short	Signed short.
Z	Boolean	True or false.

This is the standard Java encoding of the signature of a method. Thus, a method taking a Java int as arguments and returning a double could be encoded as shown here:

```
MyClass.MyMethod(I)D
```

A method taking a character and a double as an argument and returning no value could be encoded as shown here:

```
MyClass.MyMethod2(CD)V
```

... using the special encoding character V (for void) to indicate the lack of a return value.

In the standard encoding of signatures, a character encoding of a Java type can be prefixed by one or more left square brackets ([), which indicate an array of the Java type of dimensions equal to the number of brackets. As TIBCO Object Service Broker does not support array type, the only Java type that can be prefixed in this manner in a LIBNAME value is B (for byte) and then only with at most one occurrence of the bracket. A one-dimensional byte array corresponds to the characters of a TIBCO Object Service Broker string value passed to or from a Java routine.

A class type can be encoded with the letter L followed by the fully qualified class name, followed by a semicolon. The only class type supported is the Java class String, which corresponds to TIBCO Object Service Broker strings when passed to or from a Java routine.

A method taking a Java string as parameter and returning another Java string could be encoded:

```
Myclass.Mymethod(Ljava/lang/String;)Ljava/lang/String;
```

The Execution Environment uses the value of LIBNAME to instruct the JVM to load the class containing the method, locate the method within the class, and invoke the method, passing arguments to the method and returning a result (if any) to the calling TIBCO Object Service Broker rule.

The call of the routine fails if the routine specification in field LIBNAME does not conform to those rules.

Sample Entry

The single occurrence from the ROUTINES table for the routine ABC follows. In this example USERFUNJ is called as a function.

```
                        --- SINGLE OCCURRENCE EDITOR ---
EDITING TABLE    : ROUTINES
TABLE TYPE       : TDS
COMMAND ==>
--------------------------------------------------------------------------------

NAME             : USERFUNJ
LANGUAGE         : JAVA
FUNCTION         : Y
TYPE             : Q
SYNTAX           : P
LENGTH           :        5
DECIMAL          :        0
LOADNAME
SCOPE            : SESSION
NODENAME         :
                 :
                 :
                 :
                 :
LIBNAME          : MyClass.MyMoehot(I)D
PFKEYS: 1=HELP 2=DOCUMENTATION 3=SAVE 12=CANCEL 13=PRINT 22=DELETE
```

Add an Entry in the ARGUMENTS Table

If your external routine has arguments, to identify the arguments to TIBCO Object Service Broker add a table instance to the ARGUMENTS table. The parameter value for the table instance of the ARGUMENTS table must be the name of the external routine (that is, the value in the **NAME** field of the ROUTINES table). You can have a maximum of 16 arguments.

The instance of the ARGUMENTS table for the sample routine ABC follows.

```
 EDITING TABLE   :   ARGUMENTS(USERFUNJ)
COMMAND ==>
                                                                SCROLL: P
   NUMBER        NAME        INOUT TYPE SYNTAX LENGTH DECIMAL
_ ------ ----------------     -     -      -    ------ ------
_       1 COST                 N     Q      P         5      2
_       2 TAX_RATE             N     Q      P         2      2




PFKEYS: 4=INSERT 16=DELETE 5=FIND NEXT 6=CHG NEXT 18=EXCLUDE 3=SAVE 12=CANCEL
At TOP
```

Type the appropriate information in the fields of the ARGUMENTS table:

NUMBER	The position of the argument in the argument list. The positions must be sequential and start at 1. Also, it must correspond to the sequence of the encoded arguments in the corresponding LIBNAME field. Syntax: B, Length: 2
NAME	The argument name. This is used only by the call-by-name version of the TIBCO Object Service Broker CALL statement, except in the following case: The string "@THIS" has a special meaning if it appears in this field in the first occurrence. The presence of this string indicates that the Java routine is not a static method of its class, that is, it has an implied "this" argument. Syntax: C, Length: 16
INOUT	Whether the value of the argument can be changed by the external routine. Must have the value N as Java supports only call-by-value parameter passing. Syntax: C, Length: 1

TYPE	The argument semantic data type. Syntax: C, Length: 1
SYNTAX	The argument syntax. Syntax: C, Length: 1
LENGTH	The argument length. Syntax: B, Length: 2
DECIMAL	The number of digits to the right of the decimal, if any. Must contain zero. Syntax: B, Length: 2

There must be exactly one occurrence (in an instance of table ARGUMENTS corresponding to a Java routine) for each argument encoded in the field LIBNAME of the corresponding occurrence in the ROUTINES table.

The data in the occurrences of ARGUMENTS and the arguments encoded in the LIBNAME field must correspond in the following manner:

Java type encoding	Field TYPE	Field SYNTAX	Field LENGTH
B		B	1
C	S	V or C	1
D		F	8
F		F	8
I		B	4
J		B	8
S		B	2
Z	L	C	1
[B	S	V	A valid string field length
L/java/lang/String;	S	V or C	A valid string field length

Similarly, if the return value encoded is not Z, it must match the same fields in the ROUTINES occurrence describing the routine.

The call of the routine fails if the routine specification in field LIBNAME and the information in the parameter instance of ARGUMENTS do not conform to these rules.

See Also *TIBCO Object Service Broker Managing Data* for information about how to use the Table Editor.

TIBCO Object Service Broker Programming in Rules for information about valid TIBCO Object Service Broker syntax and semantic data types.

Chapter 4 Using the Interface to TIBCO Enterprise Message Service™

This chapter describes how to interface to TIBCO Enterprise Message Service (EMS).

Topics

TIBCO Object Service Broker EMS Interface

Purpose of TIBCO Enterprise Message Service

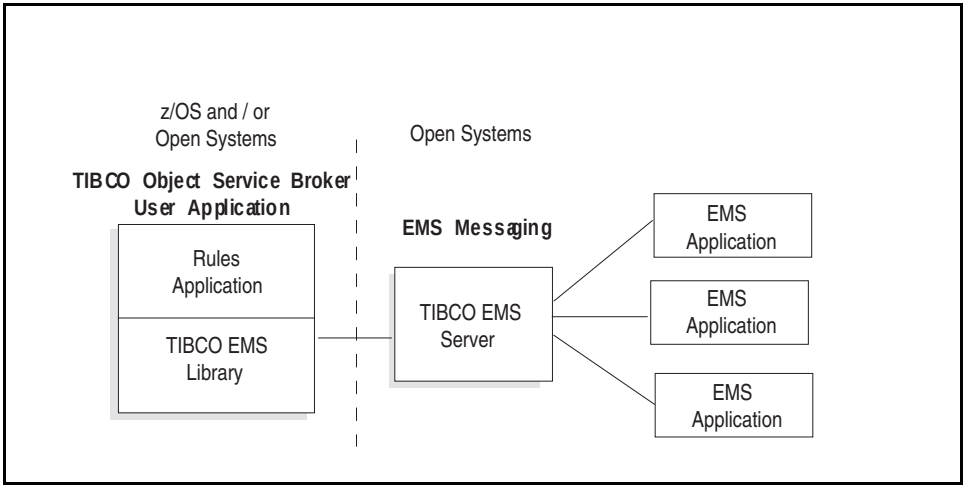
TIBCO Enterprise Message Service software lets application programs send and receive messages according to the Java Message Service (JMS) protocol. EMS is based on creation and delivery of messages. Messages are structured data that one application sends to another. The creator of the message is known as the producer and the receiver of the message is known as the consumer.

A TIBCO EMS server acts as an intermediary for the message and manages its delivery to the correct destination. The server also provides enterprise-class functionality such as fault-tolerance, message routing, and communication with other messaging systems, such as TIBCO Rendezvous™ and TIBCO SmartSockets™.

Overview of TIBCO Object Service Broker EMS Interface

The interface to EMS provides a set of tools that TIBCO Object Service Broker rules applications running on z/OS and Open Systems can use to produce and consume messages. These messages are transported via TIBCO EMS servers which run on Open Systems platforms.

This message flow is illustrated in the following diagram:



Calling EMS

Shareable Tools Available

The following shareable tools are used to interface with the TIBCO EMS Client API:

S6BCALL	Used by a rule when the EMS function does not return a value.
S6BFUNCTION	Used by a rule when a value is returned.

The types of arguments and the return value are determined by the EMS C routine being invoked. Most calls return a tibems_status value. It is possible for S6BFUNCTION to return strings or integers that are not status codes for some EMS functions.

The following is an example of a call to S6BFUNCTION:

```
STATUS = S6BFUNCTION('tibemnsMsgProducer_Send',PRODUCER,MESSAGE);
```

See Also *TIBCO Object Service Broker Shareable Tools* for details on the S6BCALL and S6BFUNCTION tools.

TIBCO Enterprise Message Service: C and COBOL Reference for the definition of the EMS API as implemented by S6BCALL and S6BFUNCTION

Argument Mapping

Mapping Data Types

C data types, as described in *TIBCO Enterprise Message Service: C and COBOL Reference,* are mapped to S6BCALL and S6BFUNCTION. Simple data types are passed as shown in the following table:

EMS C data type	S6BCALL type
tibems_byte;	Binary of length 1
tibems_short;	Binary of length 2
tibems_wchar;	Binary of length 2

EMS C data type	S6BCALL type
tibems_int;	Binary of length 4
tibems_long;	Binary of length 8
tibems_float;	Float Point of length 4
tibems_double;	Floating Point of length 8
tibems_uint;	Binary of length 4

Handles to EMS Structures

Handles to EMS structures are passed and returned as binary values of length 4. Examples of handle types include tibemsConnection, tibemsSession, and tibemsTextMsg.

Handle Management

The TIBCO Object Service Broker system is designed to handle high transaction volumes. The system therefore tracks the usage of some resources to ensure that these are not exhausted needlessly. The resources tracked include EMS connection structures, message structures, and SSL parameter structures. Whenever one of these structures has been allocated through an invocation of an EMS API function through S6BCALL or S6BFUNCTION in an TIBCO Object Service Broker transaction, terminating the transaction will implicitly release the structure, as if the TIBCO Object Service Broker application had invoked the proper EMS API function to release the structure itself. Handles to such structures may thus be used within a transaction and its child transactions, but not passed back to be used in a parent transaction.

Text Strings

In general text strings are passed as a variable character strings. In the EMS C interfaces, text strings are null terminated.

If a variable length syntax field contains a null character then the EMS interface considers that the string is terminated at that null character. Any data following will be ignored. This also holds true for UNICODE strings.

Some functions in the EMS API for C return text data using two arguments: a text area and a maximum length for the area. A rule can pass a field or a local variable for the text area. The functions are:

- tibemsDestination_GetName
- tibemsQueue_GetQueueName

Byte Oriented Data

Byte oriented data, which is typically unstructured and does not depend on an encoding, can be sent and returned through EMS using the tibemsBytes C type. S6BCALL or S6BFUNCTION arguments that refer to byte areas are defined as binary values of length 4.

Rules extract data from such areas through MAP tables. MAP areas are restricted only by job memory limits. When an EMS functions returns a tibemsBytes area then a rules program must register the area with the @MAP table before using it with a MAP table. After registering the area a rule uses the binary value for the area as a parameter for a MAP table. The parameter identifies the start of the area to be mapped by the table.

The following functions get or write tibemsBytes areas:

- tibemsBytesMsg_GetBytes
- tibemsBytesMsg_WriteBytes
- tibemsMapMsg_GetBytes
- tibemsObjectMsg_GetObjectBytes
- tibemsStreamMsg_ReadBytes
- tibemsStreamMsg_WriteBytes

See Also *TIBCO Object Service Broker Shareable Tools* for details on the @MAP table and registering MAP areas.

Error Handling

Most EMS functions return a tibems_status code if EMS detects an error. Status codes are explained in an appendix of *TIBCO Enterprise Message Service: C and COBOL Reference*.

If an abnormal termination occurs during rules processing the call to S6BCALL or S6BFUNCTION is terminated and the ROUTINEFAIL exception raised. A rules traceback is produced if the exception is not handled by the rules.

See Also *TIBCO Object Service Broker Programming in Rules* for more information about rules processing and exception.

Configuration

Initializing the EMS Interface

The only requirement to initialize the EMS interface is to ensure that the correct code page is being used. The first call to EMS by a rule initializes the environment to run EMS and loads code related to invoking EMS.

Setting the Path Environment Variable

On Windows, the EMS 6.1 or later 32-bit client portion is required for the EMS interface. The PATH environment variable must include the path to the EMS C API library DLLs, usually *TIBCO_HOME*\ems*version*\bin.

On Solaris, the LD_LIBRARY_PATH environment variable must include the path to the EMS C API library shared objects, usually *TIBCO_HOME*/ems/*version*/lib.

Code Page Support

TIBCO Object Service Broker uses a single EBCDIC code page and a single ASCII code page as defined in the @NLS1 table. Non-unicode text data is stored in this EBCDIC code page in the Data Object Broker table store. By default this code page is set to IBM-037 and the ASCII code page is set to ISO8859-1.

You use the EMSWIRECODEPAGE Execution Environment parameter to set the code page that the EMS text data is transmitted in. The default value is ISO8859-1. This wire code page can be either UTF-8, or it must match the value of the ASCII code page. The wire code page is the same for all sessions running under an Execution Environment..

The EMS function tibems_SetCodePage is not available to rules programs.

See Also

TIBCO Object Service Broker Parameters for more information about the EMSWIRECODEPAGE Execution Environment parameter.

TIBCO Object Service Broker National Language Support for more information on code pages and the @NLS1 table.

Sample Applications

Rules Samples

The @SAMPLES rules library distributed with TIBCO Object Service Broker contains a set of sample rules for using the EMS interface. Three types of sample rules are available:

1. The rules starting with S6B are generalized rules to enable the building of EMS applications.
2. The rules PUBMAPMSG, PUBMAPMSGS, PUBTEXTMSG, PUBTEXTMSGS and PUBXMLMSG are sample rules for publishing messages to EMS.
3. The rules SUBMAPMSG, SUBMAPMSGS, SUBTEXTMSG, SUBTEXTMSGS, and SUBXMLMSG are the counterpart rules that subscribe and retrieve the messages published by the publishing rules.

To use the rules listed in Sample Rules, edit the table S6BEMSURL, providing your TIBCO Object Service Broker user ID (field USERID) and the URL for the EMS server (field URL). If SSL-based message exchange is desired, combinations of the following values must also be supplied:

- If server verification is required, the server name (field SSL_HOSTNAME) and a reference to a file that contains a certificate that authenticate the server's certificate, as well as the encoding of the certificate (fields SSL_TRUSTED_PATH and SSL_TRUSTED_ENCODING). See tibemsSSLParams_AddTrustedCertFile in the EMS documentation.
- A reference to a file that contains a client certificate and its encoding (fields SSL_IDENTITY_PATH and SSL_IDENTITY_ENCODING). See tibemsSSLParams_SetIdentityFile in the EMS documentation.
- A reference to a file that contains a client private key and its encoding, if it has not been supplied as part of the client certificate (fields SSL_KEY_PATH and SSL_KEY_ENCODING). See tibemsSSLParams_SetPrivateKeyFile in the EMS documentation.
- The private key password (field SSL_PASSWORD). See tibemsConnection_CreateSSL in the EMS documentation.

Table 3 Sample Rules

Sample Rule	Function
PUBMAPMSG	Publishes the string 'Hello from TIBCO OSB' as a map message to queue TIBCO.OSB.MAPTEST.
PUBMAPMSGS	Publishes the contents of the contents of the BOOKS table as a set of map messages to the queue TIBCO.OSB.MAPTEST.
PUBTEXTMSG	Publishes the string 'Hello from TIBCO OSB' as a text message to queue TIBCO.OSB.TXTTEST.
PUBTEXTMSGS	Publishes the contents of the contents of the BOOKS table as a set of text messages to the queue TIBCO.OSB.TXTTEST.
PUBXMLMSG	Publishes the contents of the contents of the BOOKS table as a XML document to the queue TIBCO.OSB.XMLTEST.
SUBMAPMSG	Subscribes to queue TIBCO.OSB.MAPTEST and retrieves one map message and displays the contents in the message log. The counterpart to PUBMAPMSG above.
SUBMAPMSGS	Subscribes to queue TIBCO.OSB.MAPTEST and retrieves map messages and displays their contents in the message log. The counterpart to PUBMAPMSGS above.
SUBTEXTMSG	Subscribes to queue TIBCO.OSB.TXTTEST and retrieves one text message and displays the contents in the message log. The counterpart to PUBTEXTMSG above.
SUBTEXTMSGS	Subscribes to queue TIBCO.OSB.TXTTEST and retrieves text messages and displays their contents in the message log. The counterpart to PUBTEXTMSGS above.
SUBXMLMSG	Subscribes to queue TIBCO.OSB.XMLTEST and retrieves an XML document and displays its contents in tabular form in the message log. The counterpart to PUBXMLMSG above.

Supported EMS Functions

The table below lists the functions of the EMS interface for C and COBOL that are supported by TIBCO Object Service Broker. For each function that returns a handle to a newly created tracked EMS structure, the word "Tracked" appears in the Handle Action column, and the number of arguments of the function returning the handle appears in the Handle Argument column. A zero in the Handle Argument column indicates that the handle is returned as the actual value of the EMS function.

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibems_setExceptionOnFTSwitch		
tibems_GetConnectAttemptCount		
tibems_GetConnectAttemptDelay		
tibems_GetConnectAttemptTimeout		
tibems_getExceptionOnFTSwitch		
tibems_GetMulticastDaemon		
tibems_GetMulticastEnabled		
tibems_GetReconnectAttemptCount		
tibems_GetReconnectAttemptDelay		
tibems_GetReconnectAttemptTimeout		
tibems_GetSocketReceiveBufferSize		
tibems_GetSocketSendBufferSize		
tibems_IsConsumerMulticast		
tibems_Open		
tibems_SetConnectAttemptCount		
tibems_SetConnectAttemptDelay		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibems_SetConnectAttemptTimeout		
tibems_SetMulticastDaemon		
tibems_SetMulticastEnabled		
tibems_SetReconnectAttemptCount		
tibems_SetReconnectAttemptDelay		
tibems_SetReconnectAttemptTimeout		
tibems_SetSocketReceiveBufferSize		
tibems_SetSocketSendBufferSize		
tibems_SetTraceFile		
tibems_Sleep		
tibems_Version		
tibemsAdmin_Close		
tibemsAdmin_Create		
tibemsAdmin_GetCommandTimeout		
tibemsAdmin_GetConsumer		
tibemsAdmin_GetConsumers		
tibemsAdmin_GetInfo		
tibemsAdmin_GetProducerStatistics		
tibemsAdmin_GetQueue		
tibemsAdmin_GetQueues		
tibemsAdmin_GetTopic		
tibemsAdmin_GetTopics		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsAdmin_SetCommandTimeout		
tibemsBytesMsg_Create	Track	1
tibemsBytesMsg_GetBodyLength		
tibemsBytesMsg_GetBytes		
tibemsBytesMsg_ReadBoolean		
tibemsBytesMsg_ReadByte		
tibemsBytesMsg_ReadBytes		
tibemsBytesMsg_ReadChar		
tibemsBytesMsg_ReadDouble		
tibemsBytesMsg_ReadFloat		
tibemsBytesMsg_ReadInt		
tibemsBytesMsg_ReadLong		
tibemsBytesMsg_ReadShort		
tibemsBytesMsg_ReadUnsignedByte		
tibemsBytesMsg_ReadUnsignedShort		
tibemsBytesMsg_ReadUTF		
tibemsBytesMsg_Reset		
tibemsBytesMsg_SetBytes		
tibemsBytesMsg_WriteBoolean		
tibemsBytesMsg_WriteByte		
tibemsBytesMsg_WriteBytes		
tibemsBytesMsg_WriteChar		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsBytesMsg_WriteDouble		
tibemsBytesMsg_WriteFloat		
tibemsBytesMsg_WriteInt		
tibemsBytesMsg_WriteLong		
tibemsBytesMsg_WriteShort		
tibemsBytesMsg_WriteUTF		
tibemsCollection_Destroy		
tibemsCollection_GetCount		
tibemsCollection_GetFirst		
tibemsCollection_GetNext		
tibemsConnection_Close	Untrack	1
tibemsConnection_Create	Track	1
tibemsConnection_CreateSession		
tibemsConnection_CreateSSL	Track	1
tibemsConnection_GetActiveURL		
tibemsConnection_GetClientId		
tibemsConnection_GetMetaData		
tibemsConnection_IsDisconnected		
tibemsConnection_SetClientId		
tibemsConnection_Start		
tibemsConnection_Stop		
tibemsConnectionMetaData_GetEMSMajorVersion		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsConnectionMetaData_GetEMSMinorVersion		
tibemsConnectionMetaData_GetEMSProviderName		
tibemsConnectionMetaData_GetEMSVersion		
tibemsConnectionMetaData_GetProviderMajor Version		
tibemsConnectionMetaData_GetProviderMinor Version		
tibemsConnectionMetaData_GetProviderVersion		
tibemsConsumerInfo_Destroy		
tibemsConsumerInfo_GetCreateTime		
tibemsConsumerInfo_GetCurrentMsgCountSentBy Server		
tibemsConsumerInfo_GetCurrentMsgSizeSentBy Server		
tibemsConsumerInfo_GetDestinationName		
tibemsConsumerInfo_GetDestinationType		
tibemsConsumerInfo_GetDetailedStatistics		
tibemsConsumerInfo_GetDurableName		
tibemsConsumerInfo_GetElapsedSinceLast Acknowledged		
tibemsConsumerInfo_GetElapsedSinceLastSent		
tibemsConsumerInfo_GetID		
tibemsConsumerInfo_GetPendingMessageCount		
tibemsConsumerInfo_GetPendingMessageSize		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsConsumerInfo_GetStatistics		
tibemsConsumerInfo_GetTotalAcknowledgedCount		
tibemsConsumerInfo_GetTotalMsgCountSentBy Server		
tibemsConsumerInfo_IsActive		
tibemsConsumerInfo_IsConnected		
tibemsConsumerInfo_IsConnectionConsumer		
tibemsDestination_Copy		
tibemsDestination_Create		
tibemsDestination_Destroy		
tibemsDestination_GetName		
tibemsDestination_GetType		
tibemsDetailedDestStat_GetDestinationName		
tibemsDetailedDestStat_GetDestinationType		
tibemsDetailedDestStat_GetStatData		
tibemsErrorContext_Close		
tibemsErrorContext_Create		
tibemsErrorContext_GetLastErrorStackTrace		
tibemsErrorContext_GetLastErrorString		
tibemsMapMsg_Create	Track	1
tibemsMapMsg_GetBoolean		
tibemsMapMsg_GetByte		
tibemsMapMsg_GetBytes		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsMapMsg_GetChar		
tibemsMapMsg_GetDouble		
tibemsMapMsg_GetField		
tibemsMapMsg_GetFloat		
tibemsMapMsg_GetInt		
tibemsMapMsg_GetLong		
tibemsMapMsg_GetMapMsg	Track	3
tibemsMapMsg_GetMapNames		
tibemsMapMsg_GetShort		
tibemsMapMsg_GetString		
tibemsMapMsg_ItemExists		
tibemsMapMsg_SetBoolean		
tibemsMapMsg_SetByte		
tibemsMapMsg_SetBytes		
tibemsMapMsg_SetChar		
tibemsMapMsg_SetDouble		
tibemsMapMsg_SetFloat		
tibemsMapMsg_SetInt		
tibemsMapMsg_SetLong		
tibemsMapMsg_SetMapMsg		
tibemsMapMsg_SetReferencedBytes		
tibemsMapMsg_SetShort		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsMapMsg_SetStreamMsg		
tibemsMapMsg_SetString		
tibemsMsg_Acknowledge		
tibemsMsg_ClearBody		
tibemsMsg_ClearProperties		
tibemsMsg_Create	Track	1
tibemsMsg_CreateCopy	Track	2
tibemsMsg_CreateFromBytes	Track	1
tibemsMsg_Destroy	Untrack	1
tibemsMsg_GetAsBytes		
tibemsMsg_GetAsBytesCopy		
tibemsMsg_GetBodyType		
tibemsMsg_GetBooleanProperty		
tibemsMsg_GetByteProperty		
tibemsMsg_GetByteSize		
tibemsMsg_GetCorrelationID		
tibemsMsg_GetDeliveryMode		
tibemsMsg_GetDestination		
tibemsMsg_GetDoubleProperty		
tibemsMsg_GetEncoding		
tibemsMsg_GetExpiration		
tibemsMsg_GetFloatProperty		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsMsg_GetIntProperty		
tibemsMsg_GetLongProperty		
tibemsMsg_GetMessageID		
tibemsMsg_GetPriority		
tibemsMsg_GetProperty		
tibemsMsg_GetPropertyNames		
tibemsMsg_GetRedelivered		
tibemsMsg_GetReplyTo		
tibemsMsg_GetShortProperty		
tibemsMsg_GetStringProperty		
tibemsMsg_GetTimestamp		
tibemsMsg_GetType		
tibemsMsg_MakeWriteable		
tibemsMsg_Print		
tibemsMsg_PrintToBuffer		
tibemsMsg_PropertyExists		
tibemsMsg_SetBooleanProperty		
tibemsMsg_SetByteProperty		
tibemsMsg_SetCorrelationID		
tibemsMsg_SetDeliveryMode		
tibemsMsg_SetDestination		
tibemsMsg_SetDoubleProperty		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsMsg_SetEncoding		
tibemsMsg_SetExpiration		
tibemsMsg_SetFloatProperty		
tibemsMsg_SetIntProperty		
tibemsMsg_SetLongProperty		
tibemsMsg_SetMessageID		
tibemsMsg_SetPriority		
tibemsMsg_SetRedelivered		
tibemsMsg_SetReplyTo		
tibemsMsg_SetShortProperty		
tibemsMsg_SetStringProperty		
tibemsMsg_SetTimestamp		
tibemsMsg_SetType		
tibemsMsgConsumer_Close		
tibemsMsgConsumer_GetDestination		
tibemsMsgConsumer_GetMsgSelector		
tibemsMsgConsumer_GetNoLocal		
tibemsMsgConsumer_Receive		
tibemsMsgConsumer_ReceiveNoWait		
tibemsMsgConsumer_ReceiveTimeout		
tibemsMsgEnum_Destroy		
tibemsMsgEnum_GetNextName		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsMsgField_PrintToBuffer		
tibemsMsgProducer_Close		
tibemsMsgProducer_GetDeliveryMode		
tibemsMsgProducer_GetDestination		
tibemsMsgProducer_GetDisableMessageID		
tibemsMsgProducer_GetDisableMessageTimestamp		
tibemsMsgProducer_GetNPSendCheckMode		
tibemsMsgProducer_GetPriority		
tibemsMsgProducer_GetTimeToLive		
tibemsMsgProducer_Send		
tibemsMsgProducer_SendEx		
tibemsMsgProducer_SendToDestination		
tibemsMsgProducer_SendToDestinationEx		
tibemsMsgProducer_SetDeliveryMode		
tibemsMsgProducer_SetDisableMessageID		
tibemsMsgProducer_SetDisableMessageTimestamp		
tibemsMsgProducer_SetNPSendCheckMode		
tibemsMsgProducer_SetPriority		
tibemsMsgProducer_SetTimeToLive		
tibemsMsgRequestor_Close		
tibemsMsgRequestor_Create		
tibemsMsgRequestor_Request		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsObjectMsg_Create	Track	1
tibemsObjectMsg_GetObjectBytes		
tibemsObjectMsg_SetObjectBytes		
tibemsProducerInfo_Destroy		
tibemsProducerInfo_GetCreateTime		
tibemsProducerInfo_GetDestinationName		
tibemsProducerInfo_GetDestinationType		
tibemsProducerInfo_GetDetailedStatistics		
tibemsProducerInfo_GetID		
tibemsProducerInfo_GetStatistics		
tibemsQueue_Create		
tibemsQueue_Destroy		
tibemsQueue_GetQueueName		
tibemsQueueBrowser_Close		
tibemsQueueBrowser_GetMsgSelector		
tibemsQueueBrowser_GetNext		
tibemsQueueBrowser_GetQueue		
tibemsQueueInfo_Create		
tibemsQueueInfo_Destroy		
tibemsQueueInfo_GetDeliveredMessageCount		
tibemsQueueInfo_GetFlowControlMaxBytes		
tibemsQueueInfo_GetInboundStatistics		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsQueueInfo_GetMaxBytes		
tibemsQueueInfo_GetMaxMsgs		
tibemsQueueInfo_GetName		
tibemsQueueInfo_GetOutboundStatistics		
tibemsQueueInfo_GetOverflowPolicy		
tibemsQueueInfo_GetPendingMessageCount		
tibemsQueueInfo_GetPendingMessageSize		
tibemsQueueInfo_GetReceiverCount		
tibemsQueueReceiver_GetQueue		
tibemsServerInfo_Destroy		
tibemsServerInfo_GetConsumerCount		
tibemsServerInfo_GetProducerCount		
tibemsServerInfo_GetQueueCount		
tibemsServerInfo_GetTopicCount		
tibemsSession_Close		
tibemsSession_Commit		
tibemsSession_CreateBrowser		
tibemsSession_CreateBytesMessage	Track	2
tibemsSession_CreateConsumer		
tibemsSession_CreateDurableSubscriber		
tibemsSession_CreateMapMessage	Track	2
tibemsSession_CreateMessage	Track	2

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsSession_CreateProducer		
tibemsSession_CreateStreamMessage	Track	2
tibemsSession_CreateTemporaryQueue		
tibemsSession_CreateTemporaryTopic		
tibemsSession_CreateTextMessage	Track	2
tibemsSession_CreateTextMessageEx	Track	2
tibemsSession_DeleteTemporaryQueue		
tibemsSession_DeleteTemporaryTopic		
tibemsSession_GetAcknowledgeMode		
tibemsSession_GetTransacted		
tibemsSession_Recover		
tibemsSession_Rollback		
tibemsSession_Unsubscribe		
tibemsStatus_GetText		
tibemsStatData_GetByteRate		
tibemsStatData_GetMessageRate		
tibemsStatData_GetTotalBytes		
tibemsStatData_GetTotalMessages		
tibemsStreamMsg_Create	Track	1
tibemsStreamMsg_FreeField		
tibemsStreamMsg_ReadBoolean		
tibemsStreamMsg_ReadByte		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsStreamMsg_ReadBytes		
tibemsStreamMsg_ReadChar		
tibemsStreamMsg_ReadDouble		
tibemsStreamMsg_ReadField		
tibemsStreamMsg_ReadFloat		
tibemsStreamMsg_ReadInt		
tibemsStreamMsg_ReadLong		
tibemsStreamMsg_ReadShort		
tibemsStreamMsg_ReadString		
tibemsStreamMsg_Reset		
tibemsStreamMsg_WriteBoolean		
tibemsStreamMsg_WriteByte		
tibemsStreamMsg_WriteBytes		
tibemsStreamMsg_WriteChar		
tibemsStreamMsg_WriteDouble		
tibemsStreamMsg_WriteFloat		
tibemsStreamMsg_WriteInt		
tibemsStreamMsg_WriteLong		
tibemsStreamMsg_WriteMapMsg		
tibemsStreamMsg_WriteShort		
tibemsStreamMsg_WriteStreamMsg		
tibemsStreamMsg_WriteString		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsSSL_GetDebugTrace		
tibemsSSL_GetTrace		
tibemsSSL_OpenSSLVersion		
tibemsSSL_SetDebugTrace		
tibemsSSL_SetTrace		
tibemsSSLParams_AddIssuerCert		
tibemsSSLParams_AddIssuerCertFile		
tibemsSSLParams_AddTrustedCert		
tibemsSSLParams_AddTrustedCertFile		
tibemsSSLParams_Create	Track	0
tibemsSSLParams_Destroy	Untrack	1
tibemsSSLParams_GetIdentity		
tibemsSSLParams_GetPrivateKey		
tibemsSSLParams_SetAuthOnly		
tibemsSSLParams_SetCiphers		
tibemsSSLParams_SetExpectedHostName		
tibemsSSLParams_SetIdentity		
tibemsSSLParams_SetIdentityFile		
tibemsSSLParams_SetPrivateKey		
tibemsSSLParams_SetPrivateKeyFile		
tibemsSSLParams_SetRandData		
tibemsSSLParams_SetRandEGD		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsSSLParams_SetRandFile		
tibemsSSLParams_SetVerifyHost		
tibemsSSLParams_SetVerifyHostName		
tibemsTextMsg_Create	Track	1
tibemsTextMsg_GetText		
tibemsTextMsg_SetText		
tibemsTopic_Create		
tibemsTopic_Destroy		
tibemsTopic_GetTopicName		
tibemsTopicInfo_Create		
tibemsTopicInfo_Destroy		
tibemsTopicInfo_GetActiveDurableCount		
tibemsTopicInfo_GetDurableCount		
tibemsTopicInfo_GetFlowControlMaxBytes		
tibemsTopicInfo_GetInboundStatistics		
tibemsTopicInfo_GetMaxBytes		
tibemsTopicInfo_GetMaxMsgs		
tibemsTopicInfo_GetName		
tibemsTopicInfo_GetOutboundStatistics		
tibemsTopicInfo_GetOverflowPolicy		
tibemsTopicInfo_GetPendingMessageCount		
tibemsTopicInfo_GetPendingMessageSize		

Table 4 Supported EMS Functions

EMS Function	Handle Action	Handle Argument
tibemsTopicInfo_GetSubscriberCount		

Chapter 5 Using the TIBCO Service Gateway for WMQ

This chapter describes how to access IBM WebSphere MQ message queues, using the TIBCO Service Gateway for WMQ.

Topics

Overview

Service Gateway for WMQ is a Message Oriented Middleware (MOM) application containing several shared tools. You use it to create, send, receive, and process messages in a network of WebSphere MQ-enabled TIBCO Object Service Broker and non-TIBCO Object Service Broker applications. This message processing can take place across supported platforms.

For details about installing Service Gateway for WMQ, see *TIBCO Object Service Broker for Open Systems Installing and Operating*.

Service Gateway for WMQ is a separately licensed add-on to TIBCO Object Service Broker.

Configuration

Because Service Gateway for WMQ is packaged as an external routine, it must have routine and argument table entries. To create these entries, if you did not do so at installation time, make sure you are using a level-7 TIBCO Object Service Broker user ID and then run the @MOMSETUP rule.

Usage Notes

System Map Table

The interface between the rules and the WebSphere MQ software is controlled by the internal @MOMMAP map table and the corresponding MOM-specific table, for example, @MQSMAP_PORT. This is set up by the @MOMINIT shareable tool.

Required Local Variable

Prior to an @MOM... call, you must define a local variable called @MOMMAP_ADDRESS, to be available to all subsequent @MOM... calls.

WebSphere MQ Environment

Only one WebSphere MQ environment can be active in any one session at a time. The environment is owned by the transaction issuing the @MOMINIT call. The environment can be shared only with transactions executed by that transaction. You do this by passing @MOMMAP_ADDRESS.

Error Handling

The return code and reason code from WebSphere MQ are stored in the map table; refer to @MQSMAP in *TIBCO Object Service Broker Shareable Tools* for more information about this table.You can check these codes in your rules.

Example Rule

The rule that follows moves all the messages from one queue to another. In this rule, @MOMBUFFER is a MAP table set up by the writer of the rule to describe the data being written.

```
MOMPASSER;
_ LOCAL @MOMMAP_ADDRESS, CONNECTION, QUEUE1, QUEUE2;
_ ------------------------------------------------------------------------
_ ------------------------------------------------------------------+------
_ CALL @MOMINIT(1000, 'MQSERIES');                                  ¦ 1
_ CONNECTION = @MOMCONNECT('CSQ1');                                 ¦ 2
_ CALL @MOMVALIDRC;                                                 ¦ 3
_ CALL @MOMSETOPT('MQOO_INPUT_SHARED');                             ¦ 4
_ QUEUE2 = @MOMOPEN(CONNECTION, 'RON2');                            ¦ 5
_ CALL @MOMVALIDRC;                                                 ¦ 6
_ CALL @MOMSETOPT('MQOO_OUTPUT');                                   ¦ 7
_ QUEUE1 = @MOMOPEN(CONNECTION, 'RON1');                            ¦ 8
_ CALL @MOMVALIDRC;                                                 ¦ 9
_ CALL @MOMSETOPT('MQPMO_NONE');                                    ¦ A
_ @MQSMAP.GO_WAITINTERVAL = 10000000;                               ¦ B
_ @MQSMAP.GO_OPTIONS = @MOMOPTION('MQGMO_WAIT');                    ¦ C
_ @MQSMAP.GO_MATCHOPTIONS = @MOMOPTION('MQGMO_NONE');               ¦ D
_ UNTIL MOM_SHUTDOWN :                                              ¦ E
_    CALL @MOMGET(CONNECTION, QUEUE2, '@MOMBUFFER');                ¦
_    CALL @MOMVALIDRC;                                              ¦
_    CALL @MOMPUT(CONNECTION, QUEUE1, '@MOMBUFFER', 80);            ¦
_    CALL @MOMVALIDRC;                                              ¦
_    CALL @MOMCOMMIT(CONNECTION);                                   ¦
_    CALL @MOMVALIDRC;                                              ¦
_    END;                                                           ¦
_ CALL MSGLOG('SHUTDOWN RECEIVED.');                                ¦ F
_ QUEUE1 = @MOMCLOSE(CONNECTION, QUEUE1);                           ¦ G
_ CALL @MOMVALIDRC;                                                 ¦ H
_ QUEUE2 = @MOMCLOSE(CONNECTION, QUEUE2);                           ¦ I
_ CALL @MOMVALIDRC;                                                 ¦ J
_ CONNECTION = @MOMDISCONN(CONNECTION);                             ¦ K
_ CALL @MOMVALIDRC;                                                 ¦ L
_ CALL ENDMSG('NORMAL SHUTDOWN DETECTED.');                         ¦ M
_ ------------------------------------------------------------------------
_ ON MOM_INV_MOMMSG :
_    QUEUE1 = @MOMCLOSE(CONNECTION, QUEUE1);
_    QUEUE2 = @MOMCLOSE(CONNECTION, QUEUE2);
_    CONNECTION = @MOMDISCONN(CONNECTION);
_    CALL ENDMSG('INVALID MOM MSG DETECTED.');
```

See Also *TIBCO Object Service Broker Shareable Tools* about the MOM shareable tools.

Chapter 6 Using TIBCO Object Service Broker Adapter for JDBC-ODBC

This chapter describes how to use the TIBCO Object Service Broker Adapter for JDBC-ODBC.

Topics

Accessing TIBCO Object Service Broker Using 32-bit ODBC

Overview of ODBC support

What is TIBCO Object Service Broker Adapter for ODBC?

Microsoft's Open Database Connectivity (ODBC) interface is used by a wide variety of ODBC-aware applications to access data in ODBC-compliant data sources. Such DBMSs expose ODBC functionality using Structured Query Language (SQL).

The ODBC component of TIBCO Object Service Broker Adapter for JDBC-ODBC is the TIBCO Object Service Broker-specific implementation of the ODBC application programming interface (API). An application that is unaware of TIBCO Object Service Broker's specifics can use it to:

- Start and stop TIBCO Object Service Broker sessions
- Start and stop transactions within a session
- Run TIBCO Object Service Broker rules
- Request metadata related to TIBCO Object Service Broker tables and rules designated as ODBC stored procedures
- Submit SQL statements to query and modify data in TIBCO Object Service Broker tables

How Does it Work?

ODBC support for TIBCO Object Service Broker Adapter uses the OpenAccess™ product to expose the ODBC functionality to applications. OpenAccess accepts SQL statements and translates them into a standardized subset of requests passed to the TIBCO Object Service Broker-specific software layer. The latter uses the TIBCO Object Service Broker SDK (C/C++) client layer to handle TIBCO Object Service Broker sessions and transactions and to communicate with its rules-based server layer.

When an application requests an ODBC connection to TIBCO Object Service Broker, a regular TIBCO Object Service Broker session starts. The application uses standard ODBC function calls to set required connection and statement options. It submits a SQL statement that the OpenAccess layer parses and breaks into irreducible units that can be translated into the basic TIBCO Object Service Broker data requests: FORALL, INSERT, REPLACE, and DELETE. The required actions are carried out via the SDK (C/C++) and the results are returned to the OpenAccess layer.

Any number of ODBC cursors can be opened against a TIBCO Object Service Broker table simultaneously within a transaction. TIBCO Object Service Broker security settings are fully honored by the ODBC driver for TIBCO Object Service Broker when accessing tables. Because the SQL standard does not support the concept of data-parameterized tables, the ODBC driver presents a parameterized table to the ODBC caller as a flat construct, that is, data parameters are reported as fields (members of a composite key).

How Can it be Used?

Normally, third party applications make use of the ODBC driver for TIBCO Object Service Broker exactly the way they use other ODBC drivers. Users can, however, write their own applications to issue standard ODBC function calls. For details on programming environments and using ODBC, refer to relevant ODBC documentation. Refer to ODBC Conformance Levels, page 96 for details on the ODBC conformance levels supported by the TIBCO Object Service Broker Adapter for JDBC-ODBC.

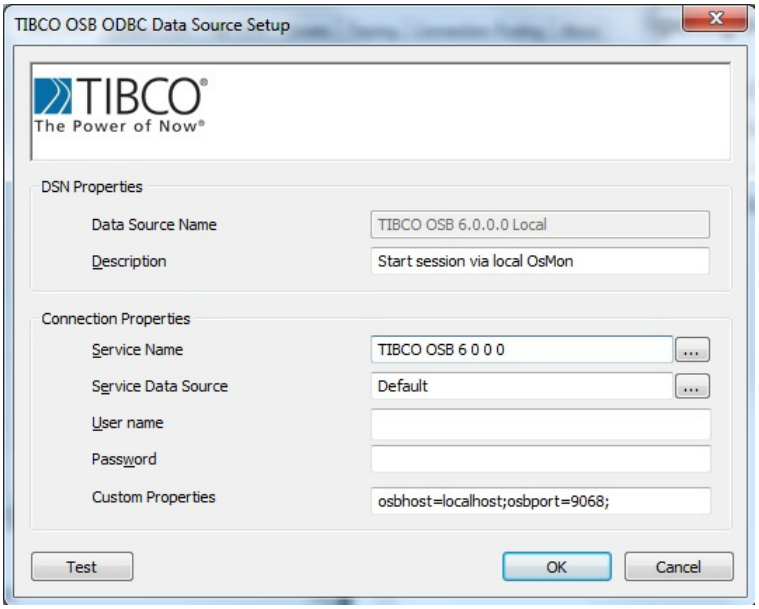
Configuring the TIBCO Object Service Broker Adapter for ODBC

The ODBC driver for TIBCO Object Service Broker uses a number of connection attributes to connect to a TIBCO Object Service Broker node. These attributes are provided via an ODBC Data Source Name (DSN) that the caller references at connection time. Alternatively, a method without DSN can be used. Refer to Connecting Without a DSN on page 81 for details on connecting without using a DSN.

Creating and Configuring a DSN for the TIBCO Object Service Broker Adapter for JDBC-ODBC

1. Open the ODBC Data Source Administrator.
2. Click the **Add...** button.
3. In the **Create New Data Source** dialog, select the TIBCO ODBC driver for the current release. For example, for the 6.0.0 release, select the **`TIBCO OSB 6.0.0.0 ODBC Adapter`**.

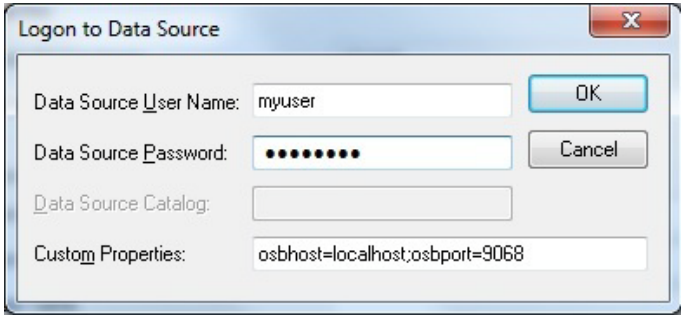
4. Click **Finish**. The TIBCO OSB ODBC Data Source Setup dialog appears. For example:



5. In the **Setup** dialog, enter a name of your data source and, optionally, a description for this data source.
6. Enter information in the **Custom Properties** field (refer to Constructing the Connect String on page 79).
7. Optionally, enter an OSB user name and password. These fields can also be entered in the Custom Properties field or not at all.
8. Make sure that correct driver is selected for the **Service Name**. The Service Name should show the TIBCO ODBC driver selected in step 3.
9. Select Default in the **Service Data Source** field.
10. Click **OK** to save the data source configuration.

Testing a Data Source

You can test a data source by pressing the **Test** button on the Setup dialog. A Logon to Data Source dialog appears, allowing you to enter a OSB user name and password and modify connection parameters.



Click the **OK** button to test whether a connection can be established for the data source.

Updating a Data Source

You can edit an existing data source by selecting a data source in the ODBC Data Source Administrator and clicking the Configure button. The dialog displayed is the same dialog used to create a data source, and described above in Creating and Configuring a DSN for the TIBCO Object Service Broker Adapter for JDBC-ODBC.

Connecting to TIBCO Object Service Broker

Applications request a connection to TIBCO Object Service Broker via a **`SQLConnect`**, a **`SQLDriverConnect`**, or a **`SQLBrowseConnect`** ODBC function call. If a DSN is referenced, the ODBC driver for TIBCO Object Service Broker assumes the session startup attributes to be available from the DSN. Therefore, it reads the Connect String from the DSN. If the caller does not reference a DSN, it is expected to provide a string that contains, along with the mandatory tokens, the same elements as the Connect String field in a DSN. Refer to Connecting Without a DSN on page 81 for details on connecting without using a DSN.

Constructing the Connect String

The value entered in the Connect String field is the actual configuration string of an ODBC client session to be started by the **`SQLConnect`**, the **`SQLDriverConnect`**, or the **`SQLBrowseConnect`** ODBC function call referring to this DSN.

The string is a sequence of keyword/value pairs, separated by semicolons. For example:

```
OSBHOST=abc;OSBPORT=9000;UID=JOHN;PWD=JOHN
```

Keyword Description

OSBHOST	A reference to the machine where the Execution Environment (osMon in case of Windows/Solaris; Native/CICS Execution Environment in case of z/OS) is running (“localhost” if connecting locally). Must be specified with PORT, if NODE is not specified. Cannot be used with NODE.
OSBPORT	The port number defined for the Execution Environment denoted by HOST. Must be specified with HOST, if NODE is not specified. Do not use with NODE.
OSBNODE	A reference to an entry in the huron.dir file describing the TIBCO Object Service Broker nodes available for connections. Cannot be used with HOST or PORT. Must be specified if HOST and PORT are not specified.
OSBSESS [optional Windows/Solaris only]	The name of a section in the session.prm file. Defaults to “DEFAULT”.
OSBEE [optional Windows/Solaris only]	The name of a section in the ee.prm file. Defaults to “DEFAULT”.
UID [optional]	The user ID for connecting to TIBCO Object Service Broker. If not available either from the caller at the connect time or from the connect string, UID defaults to the value, if any, in the respective section of the session.prm file.
PWD [optional]	The user password corresponding to UID. If not available either from the caller at the connect time or from the connect string, PWD defaults to the value, if any, in the respective section of the session.prm file.

OSBBROWSE [optional]	"Y" or "y" for TRUE; "D" or "d" for DEFAULT; any other value stands for FALSE. TRUE means that TIBCO Object Service Broker is to start up a BROWSE session. FALSE means that TIBCO Object Service Broker is to start up a NOBROWSE session. DEFAULT means that the TIBCO Object Service Broker session's BROWSE attribute is set to the value, if any, in the respective section of the session.prm file.
OSBCOML [optional]	COMMIT or ROLLBACK at COMMITLIMIT time. COMMIT means "commit updates and proceed with transaction"; ROLLBACK means "roll back changes and raise a COMMITLIMIT error condition" (standard behavior).
OSBUNIT0, OSBUNIT1, OSBUNIT2, OSBUNIT3, OSBUNIT4 [all optional]	Values of the TIBCO Object Service Broker UNIT table attribute to restrict the result set returned by the ODBC **`SQLTables`** function.
OSBCPAD [optional]	"Y" or "y" for TRUE; any other value stands for FALSE. Determines whether fixed-length character (CHAR) fields should be right- blank-padded by TIBCO Object Service Broker Adapter for JDBC-ODBC.

Connecting Without a DSN

The string supplied by way of **`SQLDriverConnect`** or **`SQLBrowseConnect`** must contain the mandatory substring:

```
DRIVER={TIBCO OSB 6.0.0.0 ODBC Adapter};PRT=TIBCO OSB 6 0 0 0
```

The mandatory substring is followed by a substring constructed according to the rules in Constructing the Connect String on page 79 and Keyword Description on page 80. For example:

```
DRIVER={TIBCO OSB 6.0.0.0 ODBC Adapter};PRT=TIBCO OSB 6 0 0
0;UID=john;PWD=JOHN;OSBSESS=efg.
```

Pre-Configured Data Sources

The installation of the TIBCO Object Service Broker Adapter for JDBC-ODBC creates two ODBC data sources with the following characteristics:

TIBCO Object Service Broker Local

Data source name	TIBCO OSB *version* Local. where *version* is the current release of the TIBCO Object Service Broker Adapter for JDBC-ODBC. For example: `TIBCO OSB 6.0.0.0 Local`
Connection string	`osbhost=localhost;osbport=9068`

TIBCO Object Service Broker Remote

Data source name	TIBCO OSB *version* Remote. where *version* is the current release of the TIBCO Object Service Broker Adapter for JDBC-ODBC. For example: `TIBCO OSB 6.0.0.0 Remote`
Connection string	`osbhost=localhost;osbport=9068`

If the TIBCO Object Service Broker Adapter for JDBC-ODBC is installed on a computer along with TIBCO Object Service Broker, the TIBCO OSB *version* Local DSN is ready for use by an ODBC-aware application. If the ODBC driver for TIBCO Object Service Broker is to start a session on another computer (for example, the TIBCO Object Service Broker Adapter for JDBC-ODBC is installed as a stand-alone client), the TIBCO Object Service Broker Remote DSN must be updated: namely, <host> should be replaced with the name or IP address of the computer where the Execution Environment runs.

These two DSNs are removed when the TIBCO Object Service Broker Adapter for JDBC-ODBC is uninstalled, and they are restored to their initial state when the TIBCO Object Service Broker Adapter for JDBC-ODBC is reinstalled. Accordingly, a new DSN should be created if it is to persist, as the installation of the TIBCO Object Service Broker Adapter for JDBC-ODBC does not affect DSNs except TIBCO OSB Local and TIBCO OSB Remote.

Configuring TIBCO Object Service Broker Components

The ODBC driver for TIBCO Object Service Broker Adapter interacts with the following TIBCO Object Service Broker components:

Component	Description
Windows/Solaris	
osMon	The TIBCO Object Service Broker Adapter for JDBC-ODBC attributes OSBHOST/OSBPORT or OSBNODE fully describe the osMon instance to be used to establish connections to TIBCO Object Service Broker.
Execution Environment	On a connection request, osMon either creates or uses an already available Execution Environment, which, in turn, creates a TIBCO Object Service Broker session. The ODBC driver connection parameters determine which Execution Environment will be used. The DOB Execution Environment parameter designates the node in the huron.dir file. This parameter can be specified either in the mon.prm or in the ee.prm file. **Note** The target Data Object Broker can reside on any of the platforms supported by TIBCO Object Service Broker.
Session	The Execution Environment creates a session for each connection requested by the ODBC driver for TIBCO Object Service Broker. If the SESS attribute is included in the connect string, its value refers to the name of a group in the session.prm file. The OSBEE, UID,PWD, and OSBBROWSE attributes take precedence, if they are part of the connect string.

Component	Description
z/OS	
Native/CICS Execution Environment	On a connection request, a standby session within an Execution Environment is allocated. A reference to a particular Execution Environment comes as part of the connection request by the ODBC driver for TIBCO Object Service Broker (OSBHOST/OSBPORT or OSBNODE). The Execution Environment's TDS parameter designates the TIBCO Object Service Broker node.
Session	The TIBCO Object Service Broker Execution Environment uses a standby session for each connection requested by the ODBC driver for TIBCO Object Service Broker.

Determining the Rules Search Path (Windows / Solaris)

As part of a session startup procedure, TIBCO Object Service Broker looks up the rules library denoted by the INSTLIB Execution Environment parameter (usually referred to as the SITE library), to pre-bind the rules available. This can constitute a performance setback for the user, especially when working with a Data Object Broker on z/OS, should the site library contain a large number of rules. This activity is mostly useless, as none of the TIBCO Object Service Broker Adapter for JDBC-ODBC's features depend on it, except for the invocation of trigger rules associated with TIBCO Object Service Broker tables.

If no trigger rules are associated with TIBCO Object Service Broker tables to be accessed by the ODBC driver for TIBCO Object Service Broker, the recommended setting for the SEARCH session parameter is S (system).

If the trigger rules to be potentially invoked can be identified in advance and moved into a separate library, specify that library as the INSTLIB Execution Environment parameter, and set the SEARCH parameter to I (installation).

Setting the COML Attribute

Normally, TIBCO Object Service Broker raises a COMMITLIMIT exception whenever its intent list becomes full. This behavior is the default for the TIBCO Object Service Broker Adapter for JDBC-ODBC as well. An ODBC application can, however, submit a SQL statement that would require a longer sequence of updates than the intent list could hold.

If the COML attribute is set to COMMIT, the TIBCO Object Service Broker Adapter for JDBC-ODBC issues a COMMIT request and proceeds with the SQL statement until completion. This allows an ODBC application to overcome the TIBCO Object Service Broker limitation on the intent list's size. This feature should be used with care, as transactional integrity is compromised by intermediate commits during what appears to be one unit of work to the caller.

Setting the BROWSE Attribute

The ODBC driver for TIBCO Object Service Broker interprets the BROWSE attribute as follows:

TRUE	All transactions are created in BROWSE mode. This means that no lock on data is taken, and no update (INSERT, UPDATE, DELETE) is carried out.
FALSE	All transactions are created in NOBROWSE mode.

The application can effectively change the BROWSE attribute via a **`SQLSetConnectOption`** function call to set the ODBC option SQL_ACCESS_MODE to either SQL_MODE_READ_WRITE or SQL_MODE_READ_ONLY.

Using the OSBUNITx Attributes

The connect string can contain the attributes OSBUNIT0, OSBUNIT1, OSBUNIT2, OSBUNIT3, and OSBUNIT4 to request the TIBCO Object Service Broker Adapter for JDBC-ODBC to restrict the result set returned by the **`SQLTables`** function. If none of these attributes is specified, no restriction applies. The restriction, if any, is effective throughout the entire session. The restriction looks similar to UNIT0=<value1> OR UNIT3=<value1>. Only those of the UNIT*x* attributes assigned values in the connect string are included in the form of a logical OR relationship.

TIBCO Object Service Broker Adapter for JDBC-ODBC and Distributed Data

The SQL-compliant notation used by the ODBC driver for TIBCO Object Service Broker for tables in SQL statements includes three dot-delimited, case-insensitive character strings:

- [Optional] Catalog, or qualifier: the only recognized value is "TIBCO". If used, schema must be specified.
- [Optional] Schema, or owner: the TIBCO Object Service Broker node name known in TIBCO Object Service Broker as location table parameter.

- Table: the TIBCO Object Service Broker table name.

Provided L is the home location, that is, the node name of the Data Object Broker to which the ODBC driver for TIBCO Object Service Broker is connected, an application can refer to the local instance of table T as follows: TIBCO.L.T, L.T, or T, whereas TIBCO.R.T and R.T refer to a remote instance of table T residing on node R. This is analogous to the TIBCO Object Service Broker native notation as in T(R).

To provide access to a table remotely, the ODBC driver for TIBCO Object Service Broker does not require that a definition of that table, whether full or minimal, be available from the local node. Also, a location parameter is not required. The following rules apply for location evaluation:

- If the location explicitly specified is different from the home location the request is directed to that location.
- If the location explicitly specified is the home location and the table name denotes:
  - A full definition, the request is processed locally
  - A minimal definition, the request is directed to the location derived from the definition (standard TIBCO Object Service Broker behavior)
- If the location is omitted and the table name denotes:
  - A full definition, the request is processed locally
  - A minimal definition, the request is directed to the location derived from the definition (standard TIBCO Object Service Broker behavior)

Locations derived from minimal definitions are honored, whereas locations derived from full definitions are ignored. This is because the SQL standard regards the notation T as identical to L.T, where L is T's home location.

Stored Procedures

Available Function Calls

The ODBC specification provides the following function calls to handle stored procedures:

Function Name	Action
SQLProcedures	Returns a result set of rows, each describing the name and properties of a stored procedure

Function Name	Action
SQLProcedureColumns	Returns a result set of rows, each describing the name and properties of a parameter of a stored procedure
SQLPrepare (SQLExecDirect)	Submits a SQL statement such as {[<r>=]call <name>[(<v1>,<v2>,…)]}, where <name> is the name of the procedure, <r> is an optional return value, and <v1>, <v2>,… are optional parameters of the procedure
SQLExecute (SQLExecDirect)	Invokes the procedure
SQLMoreResults	Switches to the next result, if any, returned by the procedure. In the discussion below, a sequence of one SQLExecute or SQLExecDirect call, followed by a series of SQLMoreResults calls on the same statement handle, is referred to as stored procedure invocation cycle.

A procedure can return the following results:

- Result set (cursor): a tabular construct to be fetched by the client
- Result count: a number indicating how many rows have been affected (-1 means “unknown”)

Designating TIBCO Object Service Broker Rules as ODBC Stored Procedures

TIBCO Object Service Broker rules to be run as ODBC stored procedure must be pre-registered in TIBCO Object Service Broker's persistent table @IP_PROCS that has three columns:

- PROCNAME—the name (for example, PROC, for the discussion below) of the rule
- ARGTAB (optional)—the name (for example, ARGT) of the table representing the procedure's parameters
- SUMMARY (optional)—the description of the procedure

If ARGTAB is an empty value, PROC is considered neither to return a value nor to have any parameters.

If ARGTAB is a non-empty value, it must be the name of TIBCO Object Service Broker table of type TEM, with the IDgen property set to Y, one data parameter defined as B 4 and one primary key (for example, ARGK) defined as I B 4. All of ARGT's fields except ARGK, in their positional order, represent PROC's optional return value and parameters. If not designated otherwise, PROC is understood to return no value and to accept INPUT parameters corresponding to all fields of ARGT, except ARGK.

The designation of a field (for example, ARGF) of table ARGT as a return value or parameter of a particular type is carried out by setting the field SOURCE of the occurrence corresponding to ARGF in the FIELDS(ARGT) table instance:

- R – return value (only one per ARGT allowed)
- I (or blank) – INPUT parameter
- O – OUTPUT parameter

Writing TIBCO Object Service Broker Rules as ODBC Stored Procedures

The following requirements apply to the rule (for example, PROC) designated as an ODBC stored procedure:

- PROC must have exactly two integer arguments (for example, HANDLE and COUNT):
  - On entry, the argument COUNT contains:

0 - the rule is being passed control for the first time within the current invocation cycle of PROC (in other words, the client has issued SQLExecute/SQLExecDirect to invoke PROC)
Positive integer (for example, N) - the rule is being passed control for the (N+1)-th time within the current invocation cycle of PROC (in other words, the client has issued SQLMoreResults for the Nth time).

  - The argument HANDLE is set by the adapter and is guaranteed to have the same value throughout the entire invocation cycle.
- PROC must not use the RETURN statement

- At the time PROC relinquishes control, it must have assigned the following locals (defined by the TIBCO Object Service Broker Adapter for JDBC-ODBC):
  - MORE_:

Y—the current invocation cycle is not over, that is, PROC expects to be invoked later via at least one more SQLMoreResults call
N (preset by the TIBCO Object Service Broker Adapter for JDBC-ODBC)—this is the last time PROC is entered within the current invocation cycle, that is, a subsequent SQLMoreResults will return SQL_DATA_NOT_FOUND

  - RESULT_:

-1—no result set (cursor) is being returned, the number of rows affected is unknown
an empty value (preset by the TIBCO Object Service Broker Adapter for JDBC-ODBC)—neither a result set (cursor) nor a result count are being returned (this is identical to returning -1)
a negative number other than -1—an error code is being returned; PROC is not to be called again within the current invocation cycle
an alphanumeric value—the name (for example, CURT) of the TIBCO Object Service Broker table representing the current cursor for the client to fetch from

The client issues an SQLExecute (or SQLExecDirect) function call, followed by a series of SQLMoreResults function calls, until it gets the SQL_DATA_NOT_FOUND return code. On the adapter side, these function calls are implemented as a respective series of invocations of PROC until it returns N in MORE_. In TIBCO Object Service Broker terms, these invocations are performed as separate events, meaning that PROC may maintain any number of any table occurrences between the calls but must reestablish the current occurrence buffers every time it is invoked.

In order to get hold of the parameter values passed by the caller, PROC is expected to issue GET ARGT(HANDLE). In order to assign the procedure's return value or OUTPUT/INPUTOUTPUT parameter values, PROC has to issue REPLACE ARGT(HANDLE). These actions can be carried out any number of times and in any sequence within an invocation cycle.

Creating Cursors in TIBCO Object Service Broker Adapter for JDBC-ODBC Stored Procedures

The TIBCO Object Service Broker Adapter for JDBC-ODBC implementation allows any number of cursors to be returned by stored procedures, meaning that no restrictions exist as to the number of times SQLMoreResults can return SQL_SUCCESS after one SQLExecute/SQLExecDirect call. This is achieved by having the procedure itself determine whether or not it is finished executing, and if a result set (cursor) is available at a particular stage.

PROC notifies the caller that a cursor is available by returning a non-empty CURT value in the RESULT_ local. This value is passed to the caller, so the ODBC fetch sequence can be carried out by the application against the cursor. This operation ends up issuing a request similar to FORALL CURT(HANDLE).

Normally, CURT refers to a table of type TEM with one data parameter defined as B 4. If so, PROC is expected to populate a cursor by issuing INSERT CURT(HANDLE). The TIBCO Object Service Broker Adapter for JDBC-ODBC clears the instance CURT(HANDLE) after it has been discarded (for example, upon a subsequent SQLMoreResults call). If, however, CURT is not of type TEM, no table clearing occurs.

If CURT has no data parameters, the FORALL request works as if FORALL CURT was issued. In other cases, an error message is returned.

Object Service Broker ODBC Stored Procedures Emulator

Prior to registering a stored procedure and driving its execution via an ODBC client, TIBCO recommends to run and debug it in the Workbench using the emulator provided as a standalone rule IP_PROC_DRV. This mode allows you to detect most of the coding errors that are difficult to fix when running the rule as a stored procedure, because rule dumps are not readily available then. Here are the tasks to follow:

1. Start the workbench.
2. Write and save your stored procedure (rule), for example PROC.
3. If the stored procedure returns a value or accepts or modifies any parameters, define a table, for example ARGT, of type TEM, with IDgen=Y and one data parameter of syntax B 4.
4. Execute rule IP_SMPL_DRV(PROC,ARGT).
5. If ARGT is a non-empty name, a table editor screen is displayed. Enter the initial values for parameters (represented by ARGT's fields).
6. Browse the message log that contains either a rule dump (in case of error) or a collection of lines representing the output produced by PROC in a context

similar to one established by the TIBCO Object Service Broker Adapter for JDBC-ODBC.

Sample

A sample rule IP _SMPL_PROC is provided with the TIBCO Object Service Broker Adapter for JDBC-ODBC:

```
IP_SMPL_PROC (HANDLE, COUNT);
_ LOCAL @WHO;
_ ------------------------------------------------------------------------
_ COUNT = 0;                                                    | Y N N N
_ COUNT = 1;                                                    |   Y N N
_ COUNT = 2;                                                    |     Y N
_ ------------------------------------------------------------+--------
_ GET @IP_SMPL_ARG(HANDLE);                                     |   1 1
_ @WHO = @IP_SMPL_ARG.WHO;                                      |   2 2
_ FORALL @IP_SMPL_DATA WHERE WHO = @WHO :                       |   3
_    @IP_SMPL_CUR.SENTENCE = @WHO || ' eats ' ||                |
_      @IP_SMPL_DATA.WHAT;                                      |
_    INSERT @IP_SMPL_CUR(HANDLE);                               |
_    END;                                                       |
_ FORALL @IP_SMPL_DATA WHERE WHO ^= @WHO :                      |     3
_    @IP_SMPL_CUR.SENTENCE = @WHO || ' does not eat ' ||        |
_      @IP_SMPL_DATA.WHAT;                                      |
_    INSERT @IP_SMPL_CUR(HANDLE);                               |
_    END;                                                       |
_ MORE_ = 'Y';                                                  | 1 4
_ RESULT_ = '@IP_SMPL_CUR';                                     |   5 4
_ MORE_ = 'N';                                                  |     5
_ ------------------------------------------------------------------------
```

When invoked as stored procedure, this rule:

- On first call, produces no results and indicates more calls expected
- On second call, returns a cursor and indicates more calls expected
- On third call, returns a cursor and marks the invocation cycle as finished

Execute IP_SMPL_DRV(IP_SMPL PROC_, @IP_SMPL_ARG); enter “man” (or “bug”) for WHO in the table editor screen and save; the message log should look similar to:

```
=== ARGUMENTS ===
1 man
Pass 0; MORE_: Y,
================
Pass 1; MORE_: Y; RESULT_: @IP_SMPL_CUR
================
1 man eats meat
2 man eats bread
3 man eats fish
Pass 2; MORE_: N; RESULT_: @IP_SMPL_CUR
================
1 man does not eat grass
2 man does not eat wood
=== ARGUMENTS ===
1 man
```

This rule is registered in @IP_PROCS as

```
PROCNAME          ARGTAB
----------------- -----------------
IP_SMPL_PROC      @IP_SMPL_ARG
```

Accordingly, executing the statement {call IP_SMPL_PROC(‘man’)} via the TIBCO Object Service Broker Adapter for JDBC-ODBC must yield same results as above.

Notes on Behavior

Supported TIBCO Object Service Broker Table Types

The following TIBCO Object Service Broker internal table types are supported:

TDS	EES	SES
TEM	SUB	PRM
EXP	IMP	VSM
ADA	DAT	DB2
IDM	IMS	SLK

The TIBCO Object Service Broker Adapter for JDBC-ODBC is oblivious of the tables of any other internal table type, even if they are present in the TIBCO Object Service Broker database. Some restrictions apply due to the nature of a particular table type, namely:

- EXP tables can only be inserted.
- PRM and IMP cannot be modified.
- EES, SES, TEM, EXP do not exhibit transactional behavior, that is, the updates are applied immediately and cannot be rolled back.
- SUB tables against tables of the types listed above, if available, exhibit the behavior of their respective base types.

TIBCO Object Service Broker internal table types are not exposed to an ODBC caller. Instead, the following ODBC table types are assigned to table definitions:

SYSTEM TABLE	All definitions whose TIBCO Object Service Broker internal table type is supported and whose AUTHOR attribute is HURON or HURON2.
VIEW	All PRM and SUB definitions that are not considered SYSTEM TYPE.
TABLE	All the remaining definitions.

See Also *TIBCO Object Service Broker Managing Data* for information about table types.

Using Parameterized Tables

As the SQL standard does not provide for the concept of data-parameterized tables, the TIBCO Object Service Broker Adapter for JDBC-ODBC presents a parameterized table to the ODBC caller as a "flat" construct, that is, data parameters are reported as fields. They are considered, in their natural order, as most-significant members of a composite primary key preceding the actual primary keys in a table definition. In their capacity as fields, data parameters can be used in field lists, WHERE clauses, and so on.

To enable the use of data parameters in a WHERE clause against a table, the TIBCO Object Service Broker Adapter for JDBC-ODBC dynamically creates against that table a PRM view that allows the full resolution of the logical criterion. This is possible only for the TDS, EES, SES, and TEM table types, as well as SUB tables against those.

For those table types that do not allow a PRM definition, the following restriction applies: if the parsing of the WHERE clause does not yield a well-defined set of data parameter instances, the request is rejected as invalid. In this context, the expression “well-defined set” means zero or more fully specified data parameter instances, for example:

- WHERE P > 0 is rejected
- WHERE P LIKE '%A' is rejected
- WHERE P=1 OR P=7 is accepted if P is the only data parameter
- WHERE P1=1 AND P2='A' is accepted if P1 and P2 are the only two data parameters

How Rows are Replaced

The TIBCO Object Service Broker Adapter for JDBC-ODBC replaces rows whenever an UPDATE <table> SET… WHERE… statement is submitted. The WHERE clause is optional but, if it is part of the statement, it can mention any field, including data parameter and/or primary keys. The SET group could require that these fields be assigned new values, for example, the following statement is completely valid: `UPDATE T SET p=0, k='a' WHERE p=1 AND k='b'`. In this example, p is assumed to be a data parameter and k, a primary key.

To carry out this UPDATE request, the TIBCO Object Service Broker Adapter for JDBC-ODBC first deletes the row denoted by the WHERE clause and then inserts a new row. This technique implies that two entries are added, transparently for the caller, to the current transaction’s intent list.

How Transactions are Handled

The TIBCO Object Service Broker Adapter for JDBC-ODBC creates a TIBCO Object Service Broker transaction upon establishing a connection to TIBCO Object Service Broker. All subsequent activities are carried out within the boundaries of this transaction, until a commit/rollback request is encountered. While ODBC applications have no means to start transactions explicitly, they do control the termination of transactions via the SQL_AUTOCOMMIT connection option setting:

- If the SQL_AUTOCOMMIT option is set to SQL_AUTOCOMMIT_ON, every successful update action triggers a transaction switch, that is, the currently running transaction is implicitly terminated and a new transaction is created.
- If the SQL_AUTOCOMMIT option is set to SQL_AUTOCOMMIT_OFF, the application can terminate the current transaction at any time via a

SQLTransact (**SQLEndTrans**) function call. A new transaction is created immediately, that is, before control is returned to the caller of this function.

The TIBCO Object Service Broker Adapter for JDBC-ODBC closes all open cursors at transaction termination time, but it preserves the state of all "prepared" statement contexts. Nested transactions are neither supported nor created implicitly.

Transactions inherit their BROWSE mode from the BROWSE attribute passed to the TIBCO Object Service Broker Adapter for JDBC-ODBC at startup time (or changed subsequently by the ODBC application). A BROWSE transaction can exist indefinitely without running short of resources, as it refrains from taking locks that can be released only at transaction end.

Support for Distributed Transactions (Windows, Solaris)

The TIBCO Object Service Broker Adapter for JDBC-ODBC provides limited support for distributed transactions that a caller initializes and manages via the Microsoft Distributed Transaction Coordinator (MS DTC) product. To exercise this functionality with a COM, OLE DB, or ODBC client, follow the specifications described in the Microsoft documentation. There is no support for XA-compliant callers, that is, the SQL_ATTR_ENLIST_IN_XA option is unsupported.

The sequence of events is as follows:

1. The caller starts a distributed transaction and requests, via MS DTC, that the TIBCO Object Service Broker Adapter for JDBC-ODBC enlist.
2. MS DTC invokes the TIBCO Object Service Broker Adapter for JDBC-ODBC's SQLSetConnectOption function with SQL_ATTR_ENLIST_IN_DTC as option notation and the transaction's handle as option value.
3. The TIBCO Object Service Broker Adapter for JDBC-ODBC creates a resource manager (RM) if one does not already exist. It marks the current connection as busy and enlists it on the transaction via the RM.

   The enlistment procedure effectively passes to MS DTC the pointers to the prepare-to-commit, commit, and rollback callbacks.
4. The caller requests MS DTC to commit or roll back the transaction.

   To commit the transaction, MS DTC invokes the prepare-to-commit callback, which, in the current implementation of the TIBCO Object Service Broker Adapter for JDBC-ODBC, always reports success. MS DTC invokes the commit callback or the rollback callback in the TIBCO Object Service Broker Adapter for JDBC-ODBC.
5. The TIBCO Object Service Broker Adapter for JDBC-ODBC schedules the action to be carried out asynchronously and returns control to MS DTC.

6. When the action is done, the TIBCO Object Service Broker Adapter for JDBC-ODBC notifies MS DTC via MS DTC's callback.
7. MS DTC invokes the TIBCO Object Service Broker Adapter for JDBC-ODBC's SQLSetConnectOption function with SQL_ATTR_ENLIST_IN_DTC as option notation and NULL as option value.
8. The TIBCO Object Service Broker Adapter for JDBC-ODBC marks the connection as free for use.

ODBC Conformance Levels

The 32-bit ODBC driver bases its ODBC Conformance Levels on those supported by OpenAccess Software's OpenAccess™. Currently, the TIBCO Object Service Broker Adapter for JDBC-ODBC uses OpenAccess 7.0. Refer to the OpenAccess manual for more information.

SQL Conformance

Level of support: Minimum+, which includes SELECT, INSERT, UPDATE, DELETE, SELECT FOR UPDATE, Expressions, Nested queries, and Scalar functions.

- The TIBCO Object Service Broker Adapter for JDBC-ODBC provides no DDL functionality such as CREATE TABLE.
- TIBCO Object Service Broker table and field names cannot be longer than 16 characters.

ODBC API Conformance

All core and Level 1 and most Level 2 Microsoft ODBC 3.51 API function calls are supported. Use the SQL GetFunctions function call to see a full list.

Error Codes and Messages

The TIBCO Object Service Broker Adapter for JDBC-ODBC follows the ODBC specification with respect to handling error situations, and returns the following to the caller of the **`SQLError`** function:

- SQLSTATE: a five-character string identifying the error situation.
- Native error code: a numeric value specific to the TIBCO Object Service Broker Adapter for JDBC-ODBC.

- Error message: a character string containing a text error message. The string is prepended with one or more component identifiers in square brackets. If the last of those identifiers is [TIBCO Object Service Broker ODBC], the message comes from the TIBCO Object Service Broker Adapter for JDBC-ODBC. Otherwise, another software layer between the caller and the TIBCO Object Service Broker Adapter for JDBC-ODBC is responsible for the message.

SQLSTATE	Native Error	Error Message
01000	-2	COMMITLIMIT has been reached.
23000	-4	NULL in non-nullable field.
S1000	-12	DEFINITION inaccessible/corrupt or unsupported TABLE type.
S1000	-16	ACCESSFAIL: check your definition.
S1000	-24	Reference check on field failed.
23000	-28	Invalid data in WHERE clause.
22005	-32	Resource locked by another session.
40001	-36	Insufficient security clearance.
42000	-40	Unrecoverable error in TIBCO Object Service Broker.
S1000	-44	External Server unavailable.
S1000	-64	Peer/External Server unavailable (or invalid TIBCO Object Service Broker NODE).
S1000	-68	SERVERFAIL.
S1000	-80	Data conversion error while handling row.
07006	-98	SDK (C/C++) general error.
S1000	-99	Software version mismatch: contact TIBCO support.
S1000	-200	Invalid table PARAMETER value(s).
S1000	-207	Error while invoking/running EVENT rule.
37000	-208	Attempt to modify a PRM- or IMP-type TABLE.

SQLSTATE	Native Error	Error Message
37000	-209	Action other than INSERT against an EXP-type table.
37000	-210	Attempt to modify an IMP-type TABLE.
37000	-211	Attempt to modify a PRM-type TABLE.
37000	-212	TRANSACTION is browse-only.
37000	-213	TABLE is read-only, or TRANSACTION is browse-only.
37000	-253	SOURCE definition inaccessible/corrupt or wrong TABLE type.
23000	-255	Failure to INSERT row.
23000	-256	Failure to REPLACE row.
S1000	-257	Failure to DELETE row.
08001	-270	Unable to create SESSION (see the logs for details).
S1000	-271	CLI Tabular Interface general error.
S1000	-272	Failure to COMMIT data.
S1001	-275	Short of memory.
S1001	-276	Too many SESSIONs.
S1090	277	NAME too long.
37000	-278	Table PARAMETER too long.
S1000	-286	Value is NULL.
S1001	-288	Value too long to fit in buffer.
01004	-289	Data right-truncated.
S1090	-1100	Unexpected DAM request (internal logic error).
S1090	-1099	Only TIBCO supported as CATALOG (QUALIFIER).
S1C00	-1098	Search pattern for CATALOG (QUALIFIER) not supported.

SQLSTATE	Native Error	Error Message
S1C00	-1097	Search pattern for SCHEMA (OWNER) not supported.
01S00	-1096	Only local SCHEMA (OWNER) supported for stored procedures.
01S00	-1095	Invalid keyword in connection string.
S1001	-1094	Too many tables requested for transaction.
S1C00	-1093	PREPARE TO COMMIT not supported.
01S02	-1092	QUALIFIER (CATALOG) reset to TIBCO.
01S02	-1091	Asynchronous processing not supported. Option reset to OFF.
37000	-1090	DATA SOURCE is read-only. Persistent data updates disallowed.
S1C00	-1089	Option value not supported.
S1010	-1088	Attempt to release HENV while sessions active.
S1011	-1087	Unable to set TXN isolation while transaction open.
23000	-1086	Table PARAMETER[s] incomplete/ambiguous.
23000	-1085	NULL in table PARAMETER.
23000	-1084	NULL in primary key.
07005	-1083	No columns to describe.
S1002	-10824	Column number greater than number of columns in result set.

For positive values of the Native Error Code, refer to Chapter 7, Using TIBCO Object Service Broker SDK (C/C++), on page 109.

Accessing TIBCO Object Service Broker Using 64-bit ODBC

Overview of 64-bit ODBC Support

The 64-bit ODBC driver for TIBCO Object Service Broker allows a 64-bit application to access and modify TIBCO Object Service Broker table data and run stored procedures. The 64-bit ODBC driver is implemented using a 3-tier architecture. A "thin" 64-bit ODBC driver communicates via TCP/IP with a SQL service that in turn passes data access request to TIBCO Object Server Broker SDK using TIBCO Object Service Broker SDK for Java. The SQL service used by the 64-bit ODBC service is the same SQL service used by the JDBC driver and literally can be shared with JDBC drivers.

The SQL service is installed automatically by the OSB SDK Client Installer.

Running the SQL Service

To use the 64-bit ODBC driver you must first start the SQL Service.

On Windows systems, the SQL Service is installed as a Windows Service and is automatically started each time you reboot your machine. The services report problems in log files located in the directory `jdbc/service/logging`. When running production systems, you should periodically clear these directories of old log files.

On UNIX systems, scripts are provided that enable a system administrator to set the SQL Service up to run as a daemon process. See Running the SQL Service on page 104.

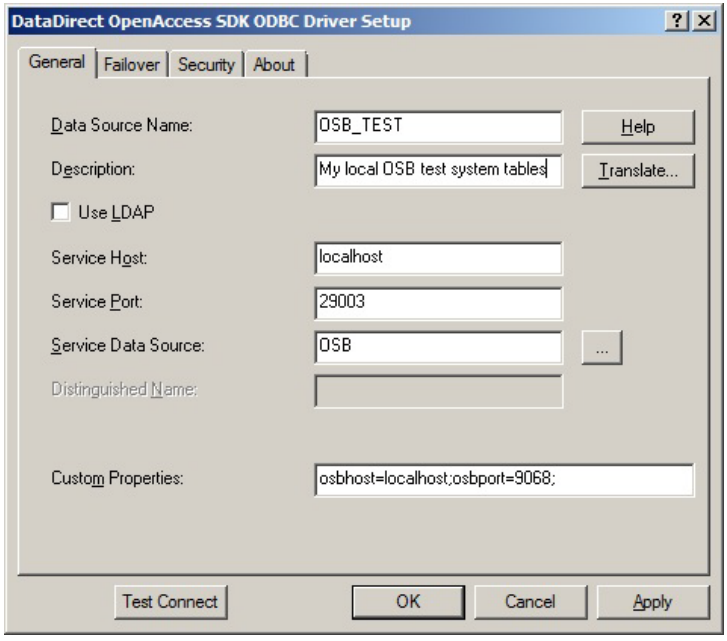
Creating and Configuring a Data Source

The ODBC driver for TIBCO Object Service Broker uses a number of connection attributes to connect to a TIBCO Object Service Broker node. These attributes are provided via an ODBC Data Source Name (DSN) that the caller references at connection time. The attributes supported by the 64-bit ODBC driver are the same as the connection parameters for the ODBC driver for 32-bit applications. Please refer to Keyword Description on page 80 for a description of the attributes.

The Microsoft ODBC Data Source Administrator is used to create ODBC data sources. However, on 64-bit Windows systems there are two versions of the administrator tool, one for 32-bit application and one for 64-bit application. For the 64-bit ODBC driver, use the Data Source Administrator that's located in the "Administrative Tools".

To create a new data source:

1. Open the ODBC Data Source Administrator from Administrative Tools menu in the control panel.
2. Click the **Add...** button.
3. In the **Create New Data Source** dialog, select the 64-bit TIBCO ODBC driver for the current release. For example, for the 6.0.0 release, select the driver named **`TIBCO OSB 6.0.0.0 ODBC 64-bit Adapter`**.
4. Click the **Finish** button. The TIBCO OSB ODBC Data Source Setup dialog appears. For example:



5. In the **Data Source Name** and **Descriptions** boxes, enter a name of your data source and, optionally, a description for this data source.
6. Set the **Service Host** name to the name of the machine where the SQL service is running. Normally this name is set to **`localhost`**.
7. Note: The **Use LDAP** check box and other tabs of the data source editor are currently not supported.
8. Set the **Service Port** to the port number of the SQL service port. By default the TIBCO client installer sets this to 19988.
9. Set the **Service Data Source** to **`OSB`**. Please note that Service Host and Service Port must be set if you wish to browse what Service Data sources are available.

10. In the **Custom Properties** field, enter the connection attributes required to connect to the TIBCO Object Service Broker node. Please refer to Keyword Description on page 80 for a description of the attributes.
11. Optionally, click on the **Test Connection** button to verify that the data source is functional.

When testing a data source please remember that the SQL service and your TIBCO Object Service Broker must be started to successfully test a data source.

12. Click **Apply** to immediately save the values you have entered during the editing processes.
13. Click **OK** to close the data source editor.

Editing an Existing Data Source

You can edit an existing data source by selecting a data source in the ODBC Data Source Administrator and clicking the Configure button. The dialog displayed is the same dialog used to create a data source.

Using the 64-bit ODBC Driver

The 64-bit ODBC, 32-bit ODBC, and the JDBC drivers share the same backend implementation. Error codes, messages, implementation of stored procedures, and table behavior are the same across the interfaces.

For details on these processes, see Accessing TIBCO Object Service Broker Using 32-bit ODBC on page 76.

Connecting Without a DSN

The string supplied by way of SQLDriverConnect or SQLBrowseConnect must contain the mandatory substring:

`DRIVER={"`*driver-name*`"};ServerDataSource=OSB;HOST=`*TIBCO-SQL-Service-host-name*`;PORT=`*TIBCO-SQL-Service-port*`;`

where :

- *driver-name* is the name of the 64-bit TIBCO OSB driver for ODBC.
- *TIBCO-SQL-Service-host-name* is the name of the host where the TIBCO SQL Service is running.
- *TIBCO-SQL-Service-port* is the port number of the TIBCO SQL Service.

The mandatory substring is followed by a substring constructed according to the rules in Constructing the Connect String and Keyword Description on page 80. For example:

```
DRIVER={"TIBCO OSB 6.0.0.0 ODBC 64bit
Adapter"};ServerDataSource=OSB;HOST=localhost;PORT=19988;OSBHOST=l
ocalhost;OSBPORT=9068;UID=HURON1;PID=HURON1;
```

ODBC Conformance Levels

The 64-bit ODBC driver bases its ODBC conformance levels on those supported by OpenAccess Software's OpenAccess™. Currently, the TIBCO Object Service Broker Adapter for JDBC-ODBC uses OpenAccess 7.0. Refer to the OpenAccess manual for more information.

Accessing TIBCO Object Service Broker Using JDBC

Overview of JDBC support

The TIBCO Object Service Broker JDBC-ODBC Adapter provides JDBC access to any Java-enabled applet, application, or application server. JDBC supports interfaces for querying and updating data and running stored procedures in a database. The JDBC driver for TIBCO Object Service Broker Adapter for JDBC-ODBC is a type 4 driver and is compliant with the JDBC 3.0 specification.

The Adapter is implemented by a 3-tier architecture. Your Java application uses a "thin" JDBC driver to access TIBCO Object Server Broker tables and rules. The "thin" client communicates via TCP/IP with a SQL service which processes SQL requests and handles joins. The SQL service runs only on Open Systems and uses the TIBCO Object Server Broker SDK for Java to access and modify Object Service Broker table data.

Running the SQL Service

To use the JDBC client driver you must first start the SQL Service.

On Windows the Service is installed as a Windows Service and can be automatically started each time you reboot your machine.

On UNIX use the script files `startservices.sh` and `stopservices.sh` in the directory `jcbc/service/admin` to start and stop the SQL Services. The services run as daemon processes, so you can close the terminal window after starting the services without terminating the services.

A UNIX system administrator can use these scripts to change system start-up so that the SQL services automatically are started and stopped each time a system is booted and shutdown. Similar scripts are available for a Windows installation.

The services report problems in log files located in the directory `jdbc/service/logging`. When running production systems, you should periodically clear these directories of old log files.

Setting the CLASSPATH

The JDBC Client must be included in your CLASSPATH variable. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate the JDBC driver on your computer. If it is not defined on your CLASSPATH, you will receive a "class not found" error when trying to load the JDBC Client. You can find the driver jar file, oajc.jar, in the jdbc/client directory of your TIBCO Object Service Broker JDBC Adapter install.

Registering the JDBC Client

To use the JDBC Client, you first must register it with the JDBC Driver Manager. That can be accomplished in one of three ways:

1. Set the Java property jdbc.drivers using the Java -D option. The jdbc.drivers property is defined as a colon-separated list of driver class names. For example:

   ```
   java -Djdbc.drivers=com.ddtek.jdbc.openaccess.OpenAccessDriver
   ```

2. Set the Java property jdbc.drivers from within your Java application or applet. To do this, include the following code in your application or applet, and call `DriverManager.getConnection()`:

   ```
   Properties p = System.getProperties();
   p.put ("jdbc.drivers",
   "com.ddtek.jdbc.openaccess.OpenAccessDriver");
   System.setProperties (p);
   ```

3. Explicitly load the driver class using the standard `Class.forName()` method. To do this, include the following code in your application or applet and call:

   ```
   DriverManager.getConnection():
   Class.forName("com.ddtek.jdbc.openaccess.OpenAccessDriver");
   ```

Specifying the JDBC Driver Connection URLs

When creating a JDBC connection you need to specify a connection URL that includes Adapter SQL Service connection information and TIBCO Object Service Broker parameters. A connection URL for the TIBCO Object Service Broker JDBC driver follows the following format:

`jdbc:openaccess://`*hostname*`:`*port*`;ServerDataSource=OSB;CustomProperties=(key=value;...)`

where:

hostname is the name of the host or the IP address where the SQL Service is running. If the JDBC driver is running on the same machine where the Adaptor SQL Service is running then LOCALHOST will suffice.

port is the TCP/IP port on which the SQL Server is listening. A default installation of the SQL Server uses the port 19988.

`ServerDataSource` must be set to **`OSB`**.

`CustomProperties` is set to the connection parameter required to connect to your Object Service Broker Execution Environment.

Note that JDBC URL parameters are separated by semicolons.

The following keywords are supported in `CustomProperties` JDBC parameter and are the same properties supported by the Object Service Broker ODBC Adapter:

Table 5 Supported JDBC Custom Properties Keywords

Keyword	Description
OSBHOST	is the reference to the machine where the Execution Environment (osMon in case of Windows/Solaris; Native/CICS Execution Environment in case of z/OS) is running ("localhost" if connecting locally). This parameter must be specified with PORT, if NODE is not specified but cannot be used with NODE.
OSBPORT	is the port number defined for the Execution Environment denoted by HOST. This parameter must be specified with HOST, if NODE is not specified but cannot be used with NODE.
OSBNODE	is a node name entry in the huron.dir file that describes the TIBCO Object Service Broker nodes available for connections. This parameter cannot be used with HOST or PORT. It must be specified if HOST and PORT are not specified.
OSBSESS	[optional Windows/Solaris only] is the name of a section in the session.prm file and defaults to "DEFAULT".
OSBEE	[optional Windows/Solaris only] is the name of a section in the ee.prm file and defaults to "DEFAULT".
OSBUID	[optional] is the user ID for connecting to TIBCO Object Service Broker. If not available OSBUID defaults to the value, if any, in the respective section of the session.prm file.
OSBPWD	[optional] is the user password corresponding to OSBUID. If OSBPWD defaults to the value, if any, in the respective section of the session.prm file.

Table 5 Supported JDBC Custom Properties Keywords

Keyword	Description
OSBBROWSE	[optional] "Y" or "y" for TRUE; "D" or "d" for DEFAULT; any other value stands for FALSE. TRUE means that TIBCO Object Service Broker is to start up a BROWSE session. FALSE means that TIBCO Object Service Broker is to start up a NOBROWSE session. DEFAULT means that the TIBCO Object Service Broker session's BROWSE attribute is set to the value, if any, in the respective section of the session.prm file.
OSBCOML	[optional] COMMIT or ROLLBACK at COMMITLIMIT time. COMMIT means "commit updates and proceed with transaction"; ROLLBACK means "roll back changes and raise a COMMITLIMIT error condition" (standard behavior).
OSBUNIT0, OSBUNIT1, OSBUNIT2, OSBUNIT3, OSBUNIT4	[all optional] are values of the TIBCO Object Service Broker UNIT table attribute to restrict the table definitions returned.
OSBCPAD	[optional] "Y" or "y" for TRUE; any other value stands for FALSE. Determines whether fixed-length character (CHAR) fields should be right- blank-padded by TIBCO Object Service

Using Stored Procedures

Object Service Broker Rules can be called as stored procedure through the JDBC driver. Such Rules must be pre-registered in TIBCO Object Service Broker's persistent table @IP_PROCS. Java client code can use the CallableStatement JDBC interface object to pass parameters and execute Rules. Please refer to the "Using TIBCO Object Service Broker ODBC Adapter" chapter and review the "Designating TIBCO Object Service Broker Rules as ODBC Stored Procedures" and the "Writing TIBCO Object Service Broker Rules as ODBC Stored Procedures" sections for more information about defining a stored procedure in the TIBCO Object Service Broker. Also review the stored procedure sample in the jdbc/samples directory.

Chapter 7 Using TIBCO Object Service Broker SDK (C/C++)

This chapter describes how to use the TIBCO Object Service Broker SDK for C and C++.

Topics

Overview

The TIBCO Object Service Broker SDK (C/C++) is an application programming interface (API) used by an application to:

- Start and stop TIBCO Object Service Broker sessions
- Start and stop transactions within a session
- Call TIBCO Object Service Broker rules within the context of a transaction

The SDK (C/C++) is installed with TIBCO Object Service Broker.

Requirements

NLS

If you are connecting to an Execution Environment on z/OS, ensure that NLS is set up, with values in @NLS1 similar to the following example, according to your environment:

```
BROWSING TABLE   :  @NLS1
COMMAND ==>
                                                                    SCROLL: P
    KEY       COMPTYPE              COMPNAME                      LOCALE_CP
---------- ---------------- ------------------------------- -------------------
_          1 SELF                                          ENGL.IBM-037
_          2 REMOTE                                        ENGL.IBM-037




PFKEYS: 1=HELP 5=FIND NEXT 9=RECALL 18=EXCLUDE 19=SHOW 13=PRINT 3=END 14=EXPAND
```

How Does It Work?

The SDK (C/C++) supplies a dataIn/dataOut commarea mechanism for unformatted binary data exchange between an application and a TIBCO Object Service Broker rule. A rule called via the SDK (C/C++) can use all the TIBCO Object Service Broker facilities except the text-presentation DISPLAY statement. To facilitate commarea binary data exchange between an application and a rule, developers can use TIBCO Object Service Broker MAP tables to process data in the dataIn commarea and to return data back to the application through the dataOut commarea.

Remote Communication

The SDK (C/C++) is a remote interface that communicates with TIBCO Object Service Broker. TIBCO Object Service Broker on all platforms supports this interface in the same way. User applications can communicate with different TIBCO Object Service Broker installations on different platforms with no change to their code. They use the SDK (C/C++) whenever they want to control a session in another computer or in another work space on the same z/OS computer.

How Can It Be Used?

With the SDK (C/C++), you write an application to manage a TIBCO Object Service Broker session using a set of subroutines to an external program. Using the SDK (C/C++) functions, you can code in whichever programming language you prefer. To make the services of TIBCO Object Service Broker available to your program, you write specific routines that make use of the SDK (C/C++) and that exert complete control over TIBCO Object Service Broker sessions. Refer to Sample Application Using the SDK (C/C++) on page 138.

Compiling and Linking

The SDK (C/C++) presents its interface to the client in an executable module (oscli.dll on Windows, and liboscli.so on Solaris) that exposes its entry points. To use the interface, an application links to the executable according to application platform and environment rules. On Windows, the oscli.lib import library is provided to link to oscli.dll. The format of the entry points is supplied in the oscli.h C header file.

Thread Safety

The SDK (C/C++) client is not thread safe at a session level. In other words, when two threads try to issue an SDK (C/C++) cliProc call on the same session area, the behavior of the second client is unpredictable.

Constants

To facilitate application development, oscli.h contains the following preprocessor definitions:

CLI_MAXRULEEXPRLEN	The maximum length of a rules call string. For more information, refer to CALLRULE – Call a Rule on page 121. The value is 514.
CLI_MAXENDMSGLEN	The maximum length of a rules end message. For more information, refer to GETENDMSG – Retrieve a Rules End Message on page 125 and to the ENDMSG shareable tool. The value is 148.

See Also

TIBCO Object Service Broker Managing Data about MAP tables.

TIBCO Object Service Broker Programming in Rules about the rules language, writing rules, and transaction processing.

TIBCO Object Service Broker Parameters about starting sessions and session Execution Environment parameters.

TIBCO Object Service Broker Shareable Tools about the ENDMSG shareable tool.

SDK (C/C++) Functions

This section contains a brief overview of the functions of the SDK (C/C++) client.

Rules Calls, Session and Transaction Management

Name	Brief description	On
cliProc	Serves as the main SDK (C/C++) entry point for processing STARTSS, STARTTR, CALLRULE, STOPTR, STOPSS, RESETSS, GETENDMSG, and SESSACTIVE requests.	page 115
cliExecTran	Performs transaction start, rules call, and transaction end as a single SDK (C/C++) call.	page 127

Code Page Setting and Error Retrieval

Name	Brief description	On
cliSetCodepage	Sets an SDK (C/C++)/SDK (Java) code page.	page 129
cliErrorReasonDescr	Retrieves the textual description of an error reason code.	page 131

Commarea Helper Functions

This group of functions facilitate dataIn and dataOut commarea processing. All the functions work with the format described in CALLRULE – Call a Rule on page 121. Generally, these functions do not validate memory pointers passed as parameters.

Name	Brief description	On
cliCommCreate	Allocates memory and formats it according to the commarea format.	page 131
cliCommCreate1	Allocates and formats a single-segment commarea.	page 132

Name	Brief description	On
cliCommDelete	Deletes a commarea created by cliCommCreate or by cliCommCreate1.	page 132
cliCommFormat	Formats memory according to the commarea format.	page 133
cliCommFormat1	Formats a single-segment commarea.	page 133
cliCommSegment	Retrieves a pointer to a commarea segment.	page 134
cliCommSegments	Retrieves the number of segments in a commarea.	page 134
cliCommSegSize	Retrieves the commarea segment size.	page 135
cliCommSize	Calculates the total commarea size.	page 135
cliCommSizeCalc	Calculates the size of a commarea for a given structure.	page 136
cliCommSizeCalc1	Calculates the size of a single-segment commarea, given the size of the segment.	page 136

C Macros

Name	Brief description	On
LLCOPY_CSTR	Copies a zero-terminated string to a string with a two-byte length prefix.	page 136
LLCOPY_MEM	Copies a string with an explicitly specified length to a string with a two-byte length prefix.	page 136
LLDECLARE	Declares a string with a two-byte length prefix.	page 137
LLSETLEN	Sets a two-byte length prefix.	page 137
LLSTR	Returns a pointer to the text part of a string that has a two-byte length prefix.	page 137
LLSTRLEN	Returns the string length from a string that has a two-byte length prefix.	page 137

See Also Appendix A, SDK (C/C++) and SDK (Java) Error Reason Codes, on page 179 for a list of error reason codes issued in relation to the SDK (C/C++).

cliProc

cliProc is the main SDK (C/C++) function. It accepts specific operation requests and the meaning of most cliProc parameters depends on the specifics of the request.

```
void cliProc(CLI_SESSION   session,
             const char    * operation,
             char          * operand,
             const char    * params,
             const void    * dataIn,
             void          * dataOut,
             char          * retData,
             int           * retCode);
```

Parameters:

session	Application-supplied session work area. The SDK (C/C++) client uses this area to store all session related internal data. For the SDK (C/C++) client to function properly, the application must not modify contents of this area.
operation	Pointer to the name of the request. Valid values are:

Operation	Refer to ...
STARTSS	STARTSS – Start a Session on page 118.
STARTTR	STARTTR – Start a Transaction on page 120.
CALLRULE	CALLRULE – Call a Rule on page 121.
STOPTR	STOPTR – Stop a Transaction on page 124.
STOPSS	STOPSS – Stop a Session on page 124.
RESETSS	RESETSS – Drop a Connection to a Session on page 125.

Operation	Refer to ...
GETENDMSG	GETENDMSG – Retrieve a Rules End Message on page 125.
SESSACTIVE	SESSACTIVE – Inquire Whether Session Is Active on page 126.

These values are in ASCII for Open Systems, and in EBCDIC for z/OS. The strings do not have to be zero-terminated. To determine what operation is requested, the SDK (C/C++) client compares the supplied request name to these names until the match is found. If the request name is not one of these, cliProc fails with CLI_INVREQUEST (2) as return code.

operand	Pointer to the operand. The meaning of this parameter varies depending on the specific request (that is, the value of the *operation* cliProc parameter, described above).
params	Pointer to the operation parameters. The meaning of this parameter varies depending on the specific request (that is, the value of the *operation* cliProc parameter, described above).
dataIn	Pointer to the dataIn commarea. Used for CALLRULE requests only.
dataOut	Pointer to the dataOut commarea. Used for CALLRULE requests only.
retData	Pointer to the memory area where the result of the operation is to be stored. The nature of the result depends on the specific request (that is, the value of the *operation* cliProc parameter, described above).
retCode	Pointer to the memory area where return code of the request is to be stored. If the request succeeds, CLI_SUCCESS (0) is returned, if the request fails, the value depends on the specific request (that is, the value of the *operation* cliProc parameter, described above).

For some values of *operation*, some of these parameter are ignored. In this case, it does not matter what the parameter contains. This is different from setting a parameter to NULL, which has a specific meaning for that parameter.

Return Value: None.

Comments All the requests accepted by cliProc are session-related. Sessions are distinguished by the *session* parameter of cliProc. *session* points to application-provided storage that the SDK (C/C++) client uses to store all the data related to a particular session. The structure of this storage is internal and the application must not modify data in storage. Type CLI_SESSION is provided to declare or allocate variables large enough to hold all internal session data.

STARTSS properly formats the *session* area and the other operations assume that the area is formatted correctly. For all cliProc calls except STARTSS, if *session* was not previously passed to STARTSS or is corrupt, the behavior of the SDK (C/C++) client is undefined. The SDK (C/C++) client checks an eyecatcher area in the *session* area and, if the eyecatcher is corrupt, the operation fails with a CLI_SESSINVALID (199) error reason code. Use this error reason code as an indication of memory misuse when debugging the application.

Generally, cliProc does not perform memory accessibility checks for pointers that the application supplies. However, there are a few exceptions from this rule, as described in the cliProc request specifications that follows.

If two calls to STARTSS with the same *session* parameter are issued one right after the other, the first session becomes inaccessible because all the internal SDK (C/C++) data for that session is lost. To avoid this, always issue a STOPSS or a RESETSS before issuing another STARTSS on the same session area.

STARTSS – Start a Session

The following table lists the cliProc parameters used by this operation:

In	operation	Points to STARTSS.
	operand	Points to the SDK (C/C++)/SDK (Java) code page for the new session. The SDK (C/C++) expects a 16-byte, blank-padded code page name. If this parameter is NULL, the SDK (C/C++) uses the code page set by the most recent cliSetCodepage call. The code page name is in ASCII. For valid values, refer to cliSetCodepage on page 129.
	params	Session parameters string. It must be prefixed by two bytes giving its length, exclusive of the length of the prefix. The string must be in the SDK (C/C++)/SDK (Java) code page specified by the cliSetCodepage call or in the STARTSS operand parameter. The endian type of the length prefix is the same as the endian type of the SDK (C/C++) client platform.
Out	session	Pointer to session storage area. The area does not have to be initialized. STARTSS formats it properly.
	retData	Points to the operation return data buffer. If the STARTSS operation succeeds, the session user ID (eight bytes, blank-padded) is copied to the buffer. If STARTSS fails, an error reason code (four bytes) is placed into the buffer.
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) and CLI_STARTSS_FAILED (4).

STARTSS starts a new TIBCO Object Service Broker session.

Use the session parameters string (*params*) to define various session behavior aspects. There are a number of parameters that are specific for the SDK (C/C++). These are (names are case insensitive):

CLIHOST	TCP/IP host name of the machine that runs the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS) where the SDK (C/C++) client is to connect.

CLIPORT	TCP/IP port number of the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS) where the SDK (C/C++) client is to connect.
CLINODE	Node name of the machine that runs the TIBCO Object Service Broker monitor process (Windows or Solaris) or the z/OS Execution Environment where the SDK (C/C++) client is to connect.
CLIENDIAN	Session endian type. This parameter affects the external representation of MAP table fields with internal syntax B and "*" external syntax. Valid entries are: BIG and LITTLE (case insensitive).
CLIMODEL	A model Execution Environment communications identifier required only when VTAM connections are used. If specified, the value must be compatible with the configuration of the VTAM installation.

CLIHOST, CLIPORT, and CLINODE are used to identify the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS) on the network. You can do this in one of these two ways:

- Using the CLIHOST parameter to specify a TCP/IP host name and the CLIPORT parameter to specify a TCP/IP port number.
- Indirectly via the CLINODE parameter.

  osMon, or the Execution Environment on z/OS, must be defined as a node in huron.dir as follows: use the *host* or *ip* node attributes to specify a TCP/IP host, and use the *port* attribute to specify the listening port.

IPC node attributes are not supported for SDK (C/C++) osMon nodes. For details on *host*, *ip*, and *port*, refer to *TIBCO Object Service Broker for Open Systems Installing and Operating*.

For more information on the CLINODE parameter, refer to *TIBCO Object Service Broker Parameters*.

Make sure that your *session* parameter string contains either the CLIHOST and CLIPORT parameters, or the CLINODE parameter. Otherwise, STARTSS fails with a CLI_INVNODE (193) error reason code.

CLIENDIAN provides a way to override the application endian type for a session. This parameter affects the external representation of MAP table fields with numeric internal syntaxes and the "*" external syntax. If CLIENDIAN is not specified, the endian type natural for the SDK (C/C++) client platform is selected. The *session* area is formatted to represent a session for subsequent cliProc calls. Even if the STARTSS call fails, the area can be used in subsequent cliProc calls (all calls except SESSACTIVE and GETENDMSG fail with a CLI_CALLOUTOFSEQ (36) error reason code). Do not call a STARTSS passing *session* area pointer that represents another active session, because the information about it is overwritten.

A STARTSS cliProc request requires a password among the session parameters. If you use external security to avoid supplying a password, use a dummy value for the PASSWORD parameter.

STARTTR – Start a Transaction

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to STARTTR.
	params	Transaction parameters string. It must be prefixed by two bytes giving its length, exclusive of the length of the prefix. The string must be in the session SDK (C/C++)/SDK (Java) code page. The endian type of the length prefix is the same as the endian type of the SDK (C/C++) client platform. If the *params* pointer is NULL, all transaction parameters are assigned based on session defaults.
Out	retData	Points to a memory area where the error reason code (four bytes) is placed. If the operation succeeds, the area stays unchanged.
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) and CLI_STARTTR_FAILED (10).

STARTTR starts a transaction within a specified session. If the session already has transactions started, STARTTR starts a child transaction.

Transaction parameters are specified in the form of a string (case insensitive):

```
BROWSE | UPDATE, TEST | NOTEST, SEARCH=S | I | L, LIBRARY=libname
```

If you omit a parameter, STARTTR uses session default value specified in the session parameter string at STARTSS time. These session defaults are set by the BROWSE, TEST, SEARCH, and LIBRARY session parameters. For more information on these, refer to *TIBCO Object Service Broker Parameters*.

If the session is already stopped or abended, STARTTR fails with a CLI_CALLOUTOFSEQ (36) error reason code.

If the maximum allowed number of transactions are already running in the session, the operation fails with TOOMANYTRANS (106) error reason code. Refer to *TIBCO Object Service Broker Parameters* for information on the TRANMAXNUM Execution Environment parameter, which sets the maximum value.

CALLRULE – Call a Rule

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to CALLRULE.
	operand	Points to the maximum length for the rules return value that is placed in the *retData* buffer, including the two-byte length prefix and the terminating zero. This parameter is an unsigned integer of length two bytes. Its endian type is the same as the endian type of the SDK (C/C++) client platform. If *operand* is NULL, no return value is stored. This parameter is In/Out. Refer to *operand* under Out.
	params	Rules call string of the following form: 'RULENAME(PARAM1, PARAM2,…,PARAMn)' It must be prefixed by a two-byte string giving its length, exclusive of the length of the prefix. The string must be in the session SDK (C/C++)/SDK (Java) code page. The endian type of the length prefix is the same as the endian type of the SDK (C/C++) client platform. The maximum allowed length of the call string (excluding the length prefix) is 514 (CLI_MAXRULEEXPRLEN in oscli.h).
	dataIn	Points to the dataIn commarea. To indicate that you do not want this commarea used, set this parameter to NULL.
	dataOut	Points to the dataOut commarea populated by a rule. To indicate that you are not using the dataOut commarea, set this parameter to NULL.

Out	operand	Length (two bytes, unsigned integer, the SDK (C/C++) client platform endian type) of the whole rules return value in textual form (length prefix and terminating zero are excluded). If the *operand* is NULL, no length is written.
	retData	Points to a memory area where the error reason code (four bytes) is to be placed if the operation fails. If the operation succeeds, the area is filled with the rules return value in textual form. The maximum number of bytes written to *retData* is passed through the *operand* parameter. The return value is in the session SDK (C/C++)/SDK (Java) code page. It is prefixed by two bytes stating its length and has a terminating zero byte at the end. The endian type of the length prefix is the same as the endian type of the SDK (C/C++) client platform.
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) and CLI_CALLRULE_FAILED (11).

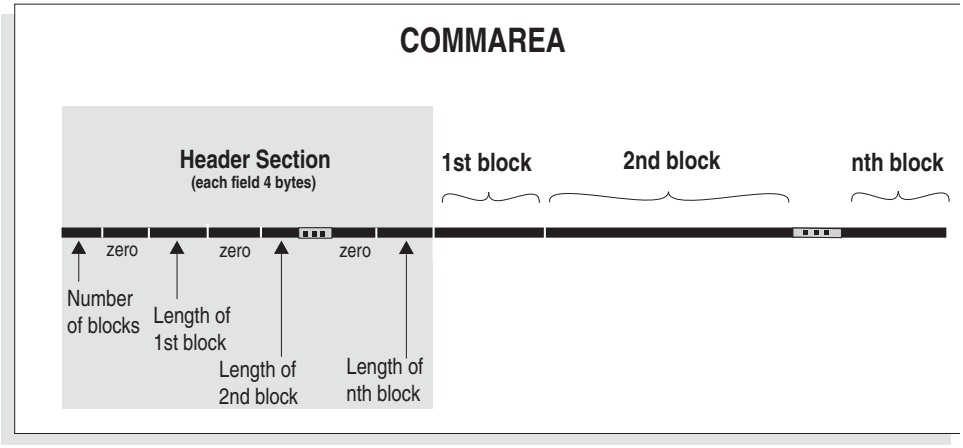
CALLRULE performs a rules call. Rules name and parameters are supplied in a textual form “RULENAME(PARAM1, PARAM2,…,PARAMn)”. The maximum length of this string is 514 (CLI_MAXRULEEXPRLEN) excluding the length prefix. If a longer string is passed, CALLRULE fails with a CLI_RULEEXPRTOOLONG (3090) error reason code. If *params* is NULL, CALLRULE fails with a CLI_NORULENAME (96) error reason code.

If the session is already abended or stopped, or no transaction is started within the session, CALLRULE fails with a CLI_CALLOUTOFSEQ (36) error reason code.

The rules return value is converted to text and placed in the memory pointed to by the *retData* parameter. The *operand* parameter is used as an In/out parameter. An application uses it to pass the number of bytes available to store the rules return value and the SDK (C/C++) client uses the parameter to return the length of the whole return value in text form, regardless of possible truncation. The returned length does not include the two-byte length prefix and terminating zero. To determine whether truncation occurred, the application can compare the resulting value to the value of the length prefix of the string in *retData* buffer.

If the rule does not return a value, an empty string is stored in *retData*. An empty string in this case is represented by three zero bytes, two for the length prefix, one for the terminating zero.

CALLRULE uses the dataIn and dataOut commareas for binary data exchange between the application and the rule. The format of a commarea is as follows:



If *dataIn* or *dataOut* does not reside in accessible memory, CALLRULE fails with the appropriate error reason code. The dataIn memory must be accessible for reading and dataOut for reading and writing.

The TIBCO Object Service Broker Execution Environment creates a copy of dataIn and makes the pointer to the area available to the rule through the APIINHANDLE field of @SESSION(0). For dataOut, the Execution Environment allocates memory for the whole area and copies the area header. A pointer to dataOut is made available through the APIOUTHANDLE field of @SESSION(0). For more information about the @SESSION table, refer to *TIBCO Object Service Broker Shareable Tools*.

Access to dataIn and dataOut using MAP tables is always granted by the system and MAP tables can be used without @MAP registration of the dataIn and dataOut addresses. dataIn is accessible for reading and dataOut for reading and writing. For more information about MAP tables, refer to *TIBCO Object Service Broker Managing Data*.

When a rule successfully completes, contents of the dataOut are transferred back from the Execution Environment to the application memory. Consider reducing the number of bytes your rule transmits to the application. You do this by properly reformatting the dataOut header, before transmitting data back, the Execution Environment reevaluates the dataOut header to determine the correct number of bytes to send back to the application.

STOPTR – Stop a Transaction

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to STOPTR.
	operand	Points to COMMIT/ROLLBACK or NULL. NULL is equivalent to COMMIT. The code page of the parameter is ASCII for Open Systems, and EBCDIC for z/OS.
Out	retData	Points to a memory area where the error reason code (four bytes) is to be placed. If the operation succeeds, the area stays unchanged.
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) and CLI_STOPTR_FAILED (12).

STOPTR commits or rolls back changes and stops the transaction active within a given session. The current transaction is destroyed and the session transaction nesting level is decremented even if STOPTR fails.

If the session is stopped, if it abended, or if no transaction is started within the session, STOPTR fails with a CLI_CALLOUTOFSEQ (36) error reason code.

STOPSS – Stop a Session

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to STOPSS.
Out	retData	Points to a memory area where the error reason code (four bytes) is to be placed. If the operation succeeds, the area stays unchanged.
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) and CLI_STOPSS_FAILED (9).

STOPSS stops the session.

If the session is already abended or stopped, STOPSS fails with a CLI_CALLOUTOFSEQ (36) error reason code.

If there are transactions active within the session, STOPSS fails with a CLI_TRANSACTIVE (128) error reason code, and the session stays active.

All other error reason codes mean that the session shutdown sequence did not complete properly (for instance, network connection was lost during the session shutdown), but the session became inactive anyway.

RESETSS – Drop a Connection to a Session

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to RESETSS.
Out	retCode	Points to a memory area where the error reason code (four bytes) is to be placed. If the operation succeeds, the area stays unchanged.

RESETSS forcefully closes the session by dropping the session connection as opposed to an orderly shutdown by STOPSS. The session does not have to be active for the call to succeed. When a connection is dropped, the Execution Environment generates an error message, and closes the session. All uncommitted data changes are lost.

If *session* is processed by STARTSS and is not modified directly by the application, RESETSS does not fail. If *session* was not previously passed to STARTSS or became corrupt, the operation can fail with CLI_SESSINVALID (199). Refer to cliProc on page 115 for information about session area validity checks.

GETENDMSG – Retrieve a Rules End Message

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to GETENDMSG.

Out	retData	Points to a memory area where the rules end message is to be placed. The end message has a two-byte length prefix and a terminating zero. The endian type of the length prefix is the same as the endian type of the SDK (C/C++) client platform. The maximum length of a TIBCO Object Service Broker rules end message is 148 (CLI_MAXENDMSGLEN in oscli.h), therefore to accommodate an end messages, the application must provide a buffer of 151 bytes. The rules end message is in the session SDK (C/C++)/SDK (Java) code page. In case of an error, a four-byte error reason code is placed in *retData*.
	retCode	Points to a buffer for the return code (four bytes). Possible values are CLI_SUCCESS (0) and CLI_GETENDMSG_FAILED (13).

GETENDMSG retrieves the most recent rules end message. The session does not have to be active for the call to succeed. If no CALLRULE was issued within the session or rules did not generate an end message, an empty string (three bytes of zeroes) is returned.

If *session* is processed by STARTSS and is not modified directly by the application, GETENDMSG does not fail. If *session* was not previously passed to STARTSS or became corrupt, the operation can fail with CLI_SESSINVALID (199). Refer to cliProc on page 115 for information about *session* area validity checks.

SESSACTIVE – Inquire Whether Session Is Active

The following table lists the cliProc parameters used by this operation:

In	session	Points to a session area. If STARTSS did not process this area, the SDK (C/C++) client behavior is undefined.
	operation	Points to SESSACTIVE.
Out	retData	Points to a memory area where the error reason code (four bytes) is to be placed in the case of an error. If the operation succeeds, a value of 1 or 0 (four-bytes, endian type of the SDK (C/C++) platform) is returned. 1 indicates that the session is still active, 0, that it is not active (either abended or stopped by STOPSS or by RESETSS).
	retCode	Points to a buffer for the return code (four bytes). Possible values of the return code are CLI_SUCCESS (0) or CLI_SESSACTIVE_FAILED (14).

It is possible for a TIBCO Object Service Broker session to become inactive any time after starting (due to network problems, Execution Environment abnormal terminations, and so on).

When that happens, STARTTR, STOPTR, CALLRULE, and STOPSS operations on this session fail with an appropriate error reason code. In addition, appropriate changes to the *session* area are made to indicate that the session is no longer active, so that subsequent STARTTR, STOPTR, CALLRULE, and STOPSS operations fail with a CLI_CALLOUTOFSEQ (36) error reason code. Use a SESSACTIVE cliProc request to determine whether the session is still active. If the session abends or is stopped by STOPSS or by RESETSS, SESSACTIVE returns 0 in the *retData* area.

If *session* is processed by STARTSS and is not modified directly by the application, SESSACTIVE should not fail. If *session* was not previously passed to STARTSS or became corrupt, the operation can fail with CLI_SESSINVALID (199). Refer to cliProc on page 115 for information about *session* area validity checks.

cliExecTran

This function combines the following cliProc actions:

- STARTTR
- CALLRULE
- STOPTR

```
void cliExecTran(CLI_SESSION      session,
                 const char       * transParam,
                 const char       * ruleName,
                 unsigned short   * retBufLen,
                 const void       * dataIn,
                 void             * dataOut,
                 char             * retData,
                 int              * retCode);
```

Parameters: In:

session	Pointer to a session area. If STARTSS did not process this area, behavior is undefined.
transParam	Transaction parameters string. For more detail, refer to STARTTR – Start a Transaction on page 120.

ruleName	Rules call in the form of the string `RULE(PARAM1, PARAM2,…PARAMn)`. For more detail, refer to CALLRULE – Call a Rule on page 121.
retBufLen	Maximum length of the *retData* buffer. For more detail, refer to CALLRULE – Call a Rule on page 121.
dataIn	dataIn commarea. For more detail, refer to CALLRULE – Call a Rule on page 121.
dataOut	dataOut commarea. For more detail, refer to CALLRULE – Call a Rule on page 121.

Out:

retBufLen	The length of the rules return value excluding the length prefix and the terminating zero. For more detail, refer to CALLRULE – Call a Rule on page 121.
retData	Pointer to the area where the rules return value is to be stored. For more detail, refer to CALLRULE – Call a Rule on page 121. If the call fails, the error reason code (four bytes) is stored in the *retData* buffer.
retCode	Pointer to the area where four bytes of a return code are to be stored. Possible values are: • CLI_SUCCESS(0) • CLI_STARTTR_FAILED(10) • CLI_CALLRULE_FAILED(11) • CLI_STOPTR_FAILED(12)

Return Value: None.

Comments If CALLRULE succeeds, STOPTR is called with the COMMIT parameter. Otherwise, STOPTR is called with the ROLLBACK parameter.

cliSetCodepage

This function indicates the SDK (C/C++)/SDK (Java) code page, that is, the code page that your application expects to use to communicate with TIBCO Object Service Broker sessions.

```
void cliSetCodepage(const char * codepage)
```

Parameter:

codepage	Address of the SDK (C/C++)/SDK (Java) code page name. The code page name is expected to be 16-byte blank-padded text. No terminating zero is required. The code page name is expected to be in ASCII. The valid values for this field are the code page names shown in *TIBCO Object Service Broker National Language Support*.

- The initial value of the code page name is IBM-037.
- WIN-1252 is recommended for use in clients that do not depend on TIBCO Object Service Broker code pages; it supports all TIBCO Object Service Broker code pages.
- ISO8859-1 works only with code pages that do not support the euro sign.
- ISO8859-15 works only with code pages that support the euro sign.

The following translations occur:

Value of code page	System specifies a non-euro code page	System specifies a euro code page
A euro code page	x'9F' (the universal currency symbol) in the non-euro code page «-» [a] x'20' in the euro code page.	

Value of code page	System specifies a non-euro code page	System specifies a euro code page
A non-euro code page		x'9F' (the universal currency symbol) in the non-euro code page «-» x'20' in the euro code page.
The Windows code page	x'80' (the euro symbol) in the Windows code page «-» x'20' in the non-euro code page. x'A4' (the universal currency symbol) in the Windows code page «-» code point x'9F' in the non-euro code page.	x'A4' (the universal currency symbol) in the Windows code page «-» x'20' in the euro code page. x'80' (the euro symbol) in the Windows code page «-» x'9F' in the non-euro code page.

a. In this table, the "«-»" symbol means "translates to, in both directions".

Return Value: None.

Comments The code page setting determines the code page of certain cliProc IN/OUT parameters as well as the external representation of MAP table fields with "*" external syntax and textual internal syntaxes.

The following cliProc parameters are affected by the setting:

- STARTSS – the session parameters string is expected in the specified code page
- STARTTR – the transaction parameter string is expected in the specified code page
- CALLRULE – the rules call expression is expected in the specified code page
- GETENDMSG – the returned end message string is in the specified code page

The code page name set by cliSetCodepage affects all sessions started after the cliSetCodepage call. Sessions that are already running are not affected.

There is a way to override this global setting on a session basis by specifying an alternative code page name as a parameter for the STARTSS operation. Refer to STARTSS – Start a Session on page 118.

cliSetCodepage stores, without validation, the code page name for future STARTSS cliProc requests that have no overriding code page specified. If the code page is not supported by TIBCO Object Service Broker, STARTSS fails with a CLI_UNSUPPCODEPAG (161) error reason code.

cliSetCodepage is thread safe and can be called at any time by any application thread. However, due to the fact that cliSetCodepage deals with the global data of the SDK (C/C++) client, some contention can occur if many threads are issuing cliSetCodepage or STARTSS cliProc requests that have no overriding code page specified, at the same time. If you need to simultaneously start sessions with varying code page settings, using the STARTSS *operand* parameter is a better choice because it does not lead to resource access synchronization by the SDK (C/C++) client.

cliErrorReasonDescr

This function retrieves a textual description of an error reason code returned by cliProc or cliExecTran.

```
const char * cliErrorReasonDescr(int reasonCode)
```

Parameter:

reasonCode	Value of the error reason code returned by cliProc or cliExecTran.

Return Value: Pointer to the textual description of the error reason code. It has a two-byte long prefix and a terminating zero.

Comments The application must not modify the contents of the description retrieved.

cliCommCreate

This function allocates memory for and formats a commarea with the given structure.

```
void * cliCommCreate(   unsigned int count,
                        unsigned int * segmentSizes)
```

Parameters:

count	Number of blocks in the commarea.
segmentSizes	Array of block sizes.

Return Value: Pointer to the beginning of the created commarea (to the *count* field of the header – refer to the commarea format description in CALLRULE – Call a Rule on page 121), or NULL if the memory allocation failed.

Comments If memory allocation fails, a NULL pointer is returned.

Segment memory is not initialized.

To delete a commarea created by cliCommCreate, you must use cliCommDelete.

cliCommCreate1

This function calls cliCommCreate to allocate memory for, and format, a one-segment commarea.

```
void * cliCommCreate1(unsigned int segmentSize)
```

Parameter:

segmentSize	Size of the only segment in the commarea.

Return Value: Pointer to the newly created area or NULL if memory allocation failed.

Comments Use cliCommDelete to delete a commarea created by this function. The segment memory is not initialized.

cliCommDelete

This function deletes a commarea created by cliCommCreate.

```
void cliCommDelete(void * area)
```

Parameter:

area	Commarea pointer returned by cliCommCreate or by cliCommCreate1.

Return Value: None.

cliCommFormat

This function formats a memory area according to the commarea format specifications, as supplied through the *count* and *segmentSizes* parameters.

```
void cliCommFormat(   void          * area,
                      unsigned int  count,
                      unsigned int  * segmentSizes)
```

Parameters:

area	Pointer to commarea memory.
count	Number of blocks in the commarea.
segmentSizes	Array of block sizes.

Return Value: None.

Comments For more about the commarea format, refer to CALLRULE – Call a Rule on page 121.

Segment memory is not initialized.

Behavior of this operation is undefined if the *area* memory area is not large enough to hold the header part of the commarea. Allocation and deallocation of the *area* memory is the responsibility of the application. Do not use cliCommDelete to deallocate the *area* memory.

cliCommFormat1

This function formats a one-segment memory area according to the commarea format specifications.

```
void cliCommFormat1(const void * area, unsigned int segmentSize)
```

Parameters:

area	Pointer to commarea memory.
segmentSize	Size of the only segment in the commarea.

Return Value: None.

Comments The commarea structure consists of one segment of *segmentSize* bytes. For details about the commarea format, refer to CALLRULE – Call a Rule on page 121.

Segment memory is not initialized.

Behavior of this operation is undefined if the *area* memory area is not large enough to hold the header part (12 bytes for the areas with one segment) of the commarea. Allocation and deallocation of the *area* memory is the responsibility of the application. Do not use cliCommDelete to deallocate the *area* memory.

cliCommSegment

This function calculates the pointer to a specific commarea segment.

```
void * cliCommSegment(const void     * area,
                      unsigned int   segmentNum)
```

Parameters:

area	Pointer to the commarea.
segmentNum	Number of a segment, starting with zero.

Return Value: Pointer to the commarea segment, or NULL if the segment does not exist (*count* field of the header is less than or equal to *segmentNum*)

Comments If the *area* memory does not comply to the commarea format rules (refer to CALLRULE – Call a Rule on page 121), the behavior is undefined.

cliCommSegments

This function retrieves the number of segments in the commarea.

```
unsigned int cliCommSegments(const void * area)
```

Parameter:

area	Pointer to the commarea.

Return Value: Number of segments in the commarea.

cliCommSegSize

This function retrieves the size of a specific commarea segment.

```
unsigned int cliCommSegSize(const void     * area,
                            unsigned int   segmentNum);
```

Parameters:

area	Pointer to the commarea.
segmentNum	Number of a segment, starting with zero.

Return Value: Size of the commarea segment based on the contents of the commarea header.

If the segment does not exist (that is, the *count* field in the header is less than or equal to *segmentNum*), this function returns 0.

Comments If the *area* memory does not comply to the commarea format rules (refer to CALLRULE – Call a Rule on page 121), the behavior is undefined.

cliCommSize

This function calculates the total size of the commarea, including the header.

```
unsigned int cliCommSize(const void * area)
```

Parameter:

area	Pointer to the commarea.

Return Value: Total size of the commarea.

Comments If the *area* memory does not comply to the commarea format rules (refer to CALLRULE – Call a Rule on page 121), the behavior is undefined.

cliCommSizeCalc

This function calculates the total number of bytes needed for a commarea with *count* blocks of sizes supplied in *segmentSizes* array.

```
unsigned int cliCommSizeCalc(unsigned int   count,
                             unsigned int   * segmentSizes)
```

Parameters:

count	Number of blocks within the commarea.
segmentSizes	Array of block sizes.

Return Value: Number of bytes needed to accommodate the commarea of the given structure.

cliCommSizeCalc1

This function calculates the total number of bytes needed for a commarea consisting of one segment of *segmentSize* bytes.

```
unsigned int cliCommSizeCalc1(unsigned int segmentSize)
```

Parameter:

segmentSize	Size of the only segment in the commarea.

Return Value: Total size needed to accommodate a commarea with one segment of *segmentSize* bytes

LLCOPY_CSTR(listr, cstr)

This C macro copies a zero-terminated string into a string with a two-byte length prefix.

LLCOPY_MEM(listr, prt, len)

This C macro copies a string with an explicitly specified length into a string with a two-byte length prefix.

LLDECLARE(name, len)

This C macro declares a string with two bytes reserved for the length prefix.

LLSETLEN(listr, len)

This C macro sets a two-byte length prefix.

LLSTR(listr)

This macro retrieves a pointer to the text part of a string that has a two-byte length prefix.

LLSTRLEN(listr)

This C macro retrieves a string length from the string's two-byte length prefix.

See Also *TIBCO Object Service Broker Managing Data* about MAP tables.

Sample Application Using the SDK (C/C++)

Compiling and Running the Sample Program

The sample program is available in the \install_folder\RemoteCLI\src folder.

In Windows

The Microsoft Visual C++ 6.0 compiler should be available. Microsoft provides a script file (*VCVARS32.BAT*) to initialize environment variables for the development environment. The following instructions assume that the OS_ROOT environment variable is initialized and the following directories exist:

- MKDIR E:\src
- MKDIR E:\obj
- MKDIR E:\bin

1. Place the source code in file E:\src\RCLISAMP.cpp
2. Compile the source code using:

```
CALL "msvc-install-bin\VCVARS32.BAT"

@REM Search in the standard TIBCO Object Service Broker
libraries for '.h' files.

@REM Search in the standard TIBCO Object Service Broker
libraries for unsatisfied external references.

@REM (The following is one command, appearing here on several
lines for readability.)

CL    /I "%OS_ROOT%\RemoteCLI\INCLUDE"
      /I "%OS_ROOT%\EXTRC\INCLUDE"
      /I "%OS_ROOT%\SRC\ENCRYPT"
      /I "%OS_ROOT%\SRC\SECURITY"
      /nologo /TP
      /Fo"E:\obj\RCLISAMP.obj"    "E:\src\RCLISAMP.cpp"
      /link /DEFAULTLIB:"%OS_ROOT%\RemoteCLI\LIB\OSCLI"
      /DEFAULTLIB:"%OS_ROOT%\EXTRC\LIB\OSEXTUSR"
      /out:"E:\bin\RCLISAMP.EXE"
```

3. Start the TIBCO Object Service Broker Data Object Broker.
4. Start the osMon.
5. Call the program using the following statement:

```
E:\bin\RCLISAMP CLIHOST=localhost,CLIPORT=9068
```

The parameter string "CLIHOST=localhost,CLIPORT=9068" contains no white space. Therefore it is passed to the program as a single parameter. The sample assumes that the PORT parameter supplied to the osMon process is defined as, or defaulted to, 9068.

In Solaris

The g++ compiler should be available in /usr/local/bin. The following instructions assume that the OS_ROOT environment variable is initialized and the following directories exist:

- mkdir $HOME/src
- mkdir $HOME/bin

1. Place the source code in file $HOME/src/RCLISAMP.cpp
2. Compile the source code using:

```
g++ -D_POSIX_C_SOURCE=199506L -U_XOPEN_SOURCE \
   -D_XOPEN_SOURCE -D__EXTENSIONS__ -D_REENTRANT \
   -I$OS_ROOT/RemoteCLI/include \
   -L$OS_ROOT/sharedlib \
   -loscli -losextusr -losscalar -loscs -losdobext \
   -losmisc -losmsgs -losencryp -lossecur -lsocket \
   -losbrand -lnsl -lpthread \
   -lrt \
   -o$HOME/bin/RCLISAMP \
   $HOME/src/RCLISAMP.cpp
```

3. Start the TIBCO Object Service Broker Data Object Broker.
4. Start the osMon.
5. Call the program using the following statement:

```
$HOME/bin/RCLISAMP CLIHOST=localhost,CLIPORT=8302
```

The parameter string "CLIHOST=localhost,CLIPORT=8032" contains no white space. Therefore it is passed to the program as a single parameter. The sample assumes that the PORT parameter supplied to the osMon process is defined as 8032.

The sample uses the HURON1 TIBCO Object Service Broker user with a password of HURON1.

Rule Called by C Program

This is a sample of a rule that creates an occurrence of the LOG TDS table, generates an end message, and returns a value. On completion of the rule, the changes are not committed because the transaction is still active. The SDK (C/C++) program explicitly stops the transaction by issuing STOPTR with a COMMIT flag or a ROLLBACK flag to indicate whether the changes are to be committed.

```
    RULE EDITOR ===>                                                SCROLL: P
TC007113RU02;
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ TC007113TA01.TEXT = 'RULE "TC007113RU02" IS CALLED';        ¦ 1
_ INSERT TC007113TA01;                                        ¦ 2
_ CALL ENDMSG('END MESSAGE GENERATED BY RULE "TC007113RU02"');¦ 3
_ RETURN('RETURN VALUE OF RULE "TC007113RU02"');              ¦ 4
_ ------------------------------------------------------------------------------
_
```

Table Referenced by a Rule

The table TC007113TA01 is defined as follows:

```
COMMAND==>                          TABLE DEFINITION

      Table: TC007113TA01     Type: TDS   Unit: TC07113              IDgen: Y
     Source:
   Parameter Name  Typ Syn Len Dec Class             '      Event Rule   Typ Acc
  ----------------  -   -  --- --    -              '   ---------------- -   -
_ LOCATION          I   C   16  0    L              ' _
_                                                   ' _
     Field Name   Typ Syn Len Dec Key Ord Rqd    Default         Reference
  ---------------- -   -  ---- --  -   -   - ---------------- ----------------
_ KEY              I   B     4  0  P
_ TEXT             S   C    50  0
_
_
_
_
_
_
_
_
_
_
PFKEYS: 3=END 12=CANCEL 22=DELETE 13=PRINT 14=FIELDS 21=DATA 2=DOC
New table definition
```

Output from C Program

The output from the C program is as follows:

```
STARTSS completed. Session User ID = HURON1
STARTTR completed.
CALLRULE completed, return value: 'RETURN VALUE OF RULE
"TC007113RU002"'
Rule end message: 'END MESSAGE GENERATED BY RULE "TC007113RU002"'
STOPTR completed.
STOPSS completed.
```

Chapter 8 Using TIBCO Object Service Broker SDK (Java)

This chapter describes how to use the TIBCO Object Service Broker SDK for Java.

Topics

Overview

The TIBCO Object Service Broker SDK (Java) is an application programming interface (API) used by an application in a Java environment to:

- Start and stop TIBCO Object Service Broker sessions
- Start and stop transactions within a session
- Call TIBCO Object Service Broker rules within the context of a transaction

It is a platform-independent version of the SDK (C/C++) described in Chapter 7, Using TIBCO Object Service Broker SDK (C/C++), on page 109.

Requirements

NLS

If you are connecting to an Execution Environment on z/OS, ensure that NLS is set up, with values in @NLS1 similar to the following example, according to your environment:

```
BROWSING TABLE   :  @NLS1
COMMAND ==>
                                                                    SCROLL: P
    KEY       COMPTYPE             COMPNAME                    LOCALE_CP
----------- ---------------- ------------------------------- ------------------
_          1 SELF                                          ENGL.IBM-037
_          2 REMOTE                                        ENGL.IBM-037
```

```
PFKEYS: 1=HELP 5=FIND NEXT 9=RECALL 18=EXCLUDE 19=SHOW 13=PRINT 3=END 14=EXPAND
```

How Does It Work?

The SDK (Java) supplies a dataIn/dataOut commarea mechanism for unformatted binary data exchange between an application and a TIBCO Object Service Broker rule. A rule called via the SDK (Java) can use all the TIBCO Object Service Broker facilities except the text-presentation DISPLAY statement. To facilitate commarea binary data exchange between an application and a rule, developers can use TIBCO Object Service Broker MAP tables to process data in the dataIn commarea and to return data back to the application through the dataOut commarea.

Remote Communication

The SDK (Java) is a remote interface that communicates with TIBCO Object Service Broker. TIBCO Object Service Broker supports this interface in the same way on all platforms with no user application code change. In a Java environment, applications use the SDK (Java) to control a session in the local or another computer (on any platform), or in another work space on the same computer (in z/OS only).

How Can It Be Used?

With the SDK (Java), you write an application to manage a TIBCO Object Service Broker session using a set of Java classes. To make the services of TIBCO Object Service Broker available to your program, you write specific code that makes use of the SDK (Java) classes and that exerts complete control over TIBCO Object Service Broker sessions. Refer to Sample Application Using the SDK (Java) on page 175.

Compiling

The SDK (Java) is supplied as a cli.jar file. To use the interface, an application calls the methods of the SDK (Java) classes. The classes within cli.jar can be made accessible to the application via the CLASSPATH system environment variable or can be embedded in your application .jar file.

Thread Safety

Most Session class methods are thread safe at a session level. In other words, when two threads try to run a method of the same SDK (Java) Session object, the behavior is unpredictable. This applies to the following: start, stop, reset, startTrans, stopTrans, call, shutdown, and execTran.

transNestLevel, endMessage, isActive, and userId can be run at the same time as any other method on the same Session object.

The methods of the SessionException and Misc classes are fully thread safe.

Constants

To facilitate application development, the Session class contains the following constants, which are defined as *static public final* fields:

MAXRULEEXPRLEN	The maximum length of a rules call string. For more information, refer to call on page 152. The value is 514.
MAXENDMSGLEN	The maximum length of a rules end message. For more information, refer to endMessage on page 155 and to the ENDMSG shareable tool. The value is 148.

See Also

TIBCO Object Service Broker Managing Data about MAP tables.

TIBCO Object Service Broker Programming in Rules about the rules language, writing rules, and transaction processing.

TIBCO Object Service Broker Parameters about starting sessions and about session Execution Environment parameters.

TIBCO Object Service Broker Shareable Tools about the ENDMSG shareable tool.

SDK (Java) Methods

Classes

The SDK (Java) client comprises a set of Java classes that reside in the com.Amdahl.Cli package:

Session	A class representing an SDK (Java) session context, and providing methods for session start up and stop, transaction start up and stop, and rules invocation.
SessionException	An exception class used to indicate Session method errors.
Misc	A class providing commarea helper functions and functions for converting numeric types to and from big-endian byte array representation.

The tables are followed by detailed information about each SDK (Java) method.

Session Class

Method	Brief description	On
Session	The class constructor.	page 150
start	Starts an SDK (Java) session.	page 159
stop	Stops an SDK (Java) session.	page 162
reset	Drops a connection to a session.	page 158
startTrans	Starts a transaction.	page 161
stopTrans	Stops the currently active transaction, committing or rolling back the changes.	page 163
call	Calls a rule.	page 152
transNestLevel	Returns the transaction nesting level of the session.	page 163
isActive	Returns the activity status of the session.	page 158

Method	Brief description	On
endMessage	Returns the end message from the last rule called within the session.	page 155
userId	Returns the user ID of the session.	page 164
shutdown	Stops all transactions (committing or rolling back the changes), and stops the session regardless of the errors encountered.	page 159
execTran	Equivalent to executing a sequence of startTrans, call, and stopTrans methods.	page 156

SessionException Class

Name	Brief description	On
SessionException	The SessionException class constructor.	page 165
rc	Returns an SDK (Java) operation error code.	page 167
reasonCode	Returns an error reason code for an SDK (Java) error.	page 166
errorReasonDescr	Returns a textual description of a particular SDK (Java) error reason code.	page 166

Refer to Appendix A, SDK (C/C++) and SDK (Java) Error Reason Codes, on page 179 for a list of error reason codes issued by the SDK (Java).

Misc Class

These functions facilitate conversions between numeric types and their big-endian representations in byte arrays. These functions do not validate input parameters.

Name	Brief description	On
readShort	Reads 2 bytes from a byte array and returns a value of type short according to what these bytes represent in big-endian format.	page 172
writeShort	Writes, into a byte array, 2 bytes of a big-endian byte representation of a value of type short.	page 173

Name	Brief description	On
readInt	Reads 4 bytes from a byte array and returns a value of type int according to what these bytes represent in big-endian format.	page 172
writeInt	Writes, into a byte array, 4 bytes of a big-endian byte representation of a value of type int.	page 173

This group of functions facilitate dataIn and dataOut commarea processing. These functions work with commareas of the format described in call on page 152. Generally, these functions do not validate input parameters.

Name	Brief description	On
commSizeCalc	Calculates the number of bytes needed for a commarea with the specified structure.	page 171
commCreate	Creates a new byte array and formats it according to the commarea specification.	page 168
commFormat	Formats a byte array according to the commarea specification.	page 169
commSegmentInd	Returns the offset of a specified segment in a commarea.	page 169
commSegSize	Returns the size of a given commarea segment.	page 170
commSize	Calculates the number of bytes in a commarea according to its header.	page 171
commSegments	Returns the number of segments in a commarea according to its header.	page 170

Session Object Methods

Session

The class constructor.

```
public Session()
```

or

```
public Session(String sessParam,
               String codepage) throws SessionException
```

Parameters:

sessParam	The session parameter string.
codepage	The SDK (C/C++)/SDK (Java) code page to be used for this session. The valid values for this field are the code page names shown in *TIBCO Object Service Broker National Language Support*.

- The initial value of the code page name is IBM-037.
- WIN-1252 is recommended for use in clients that do not depend on TIBCO Object Service Broker code pages; it supports all TIBCO Object Service Broker code pages.
- ISO8859-1 works only with code pages that do not support the euro sign.
- ISO8859-15 works only with code pages that support the euro sign.

The following translations occur:

Value of code page	System specifies a non-euro code page	System specifies a euro code page
A euro code page	x'9F' (the universal currency symbol) in the non-euro code page «-» [a] x'20' in the euro code page.	
A non-euro code page		x'9F' (the universal currency symbol) in the non-euro code page «-» x'20' in the euro code page.
The Windows code page	x'80' (the euro symbol) in the Windows code page «-» x'20' in the non-euro code page. x'A4' (the universal currency symbol) in the Windows code page «-» code point x'9F' in the non-euro code page.	x'A4' (the universal currency symbol) in the Windows code page «-» x'20' in the euro code page. x'80' (the euro symbol) in the Windows code page «-» x'9F' in the non-euro code page.

a. In this table, the "«-»" symbol means "translates to, in both directions".

For the second form of this method, these parameters are used to start a session.

Return Value: None.

Throws: The second form of this method throws SessionException with rc = SessionException.STARTSS_FAILED (4) if a session cannot be started.

Comments An application program can use the first form of the Session method, which constructs an object representing an inactive session, to create an object in preparation for starting a session later. The second form also invokes the start method to start a TIBCO Object Service Broker session.

call

Calls a rule.

```
public String call(String func,
                   byte[] dataIn,
                   byte[] dataOut) throws SessionException
```

or

```
public int    call(String func,
                   byte[] dataIn,
                   byte[] dataOut,
                   byte[] ruleRetValue,
                   int    ruleRetValueStart,
                  int   ruleRetValueMaxLen) throws SessionException
```

Parameters:

func	The rules functional expression in a textual form: `RULENAME(ARG1,ARG2,...,ARGN)`. The expression length must be less than or equal to 514 (Session.MAXRULEEXPRLEN constant).
dataIn	The input commarea: a byte array, formatted in accordance with the commarea format specification (refer to Comments on page 153).
dataOut	The output commarea: a byte array, formatted in accordance with the commarea format specification.
ruleRetValue	A byte array meant to hold the rules return value. If *ruleRetValue* is null, no return value is stored.
ruleRetValue Start	The index where the SDK (Java) should start storing the return value in the *ruleRetValue* array.
ruleRetValue MaxLen	The maximum number of bytes available for the rules return value in the *ruleRetValue* array. If *ruleRetValueMaxLen* is zero, no return value is stored.

Return Value: The first form of the method returns the rules return value as a String object.

The second form returns the actual length of the rules return value as returned by the rule. Compare this length to the value passed as *ruleRetValueMaxLen* to determine if truncation took place.

The rules return value is the value that the called rule returns in a RETURN statement.

Exceptions: SessionException with rc = SessionException.CALLRULE_FAILED (11) is thrown in case of failure.

Comments If the *func* parameter value is NULL, both forms of the call method throw an exception with an error reason code of SessionException.NORULENAME (96). Also, if the *func* parameter value is longer than 514 (Session.MAXRULEEXPRLEN), both forms throw an exception with an error reason code of Session.RULEEXPRTOOLONG (3090).

If the session already abends or is already stopped, or no transaction is started within the session, the call method throws an exception with an error reason code of SessionException.CALLOUTOFSEQ (36).

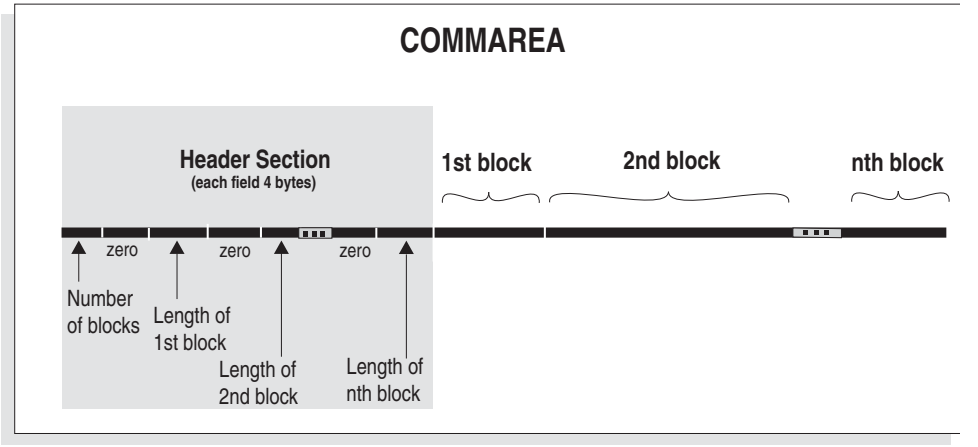
The two forms of the call method differ in how they handle the rules return value:

- The first form returns the rules return value as a newly created String object.
- The second form stores the value in the input *ruleRetValue* byte array. The value is stored in the SDK (C/C++)/SDK (Java) code page (refer to start on page 159), starting from the *ruleRetValueStart* index, for up to *ruleRetValueMaxLen* bytes. This form returns the full length of the rules return value. Therefore, if this full-length value is greater than the value passed in *ruleRetValueMaxLen*, truncation occurred.

Passing null as ruleRetValue or zero as ruleRetValueMaxLen results in no rules return value stored. This is not an error.

If the rule does not return a value, the first form of the call method returns an empty string and the second form stores no bytes in *ruleRetValue*.

Both forms of the call method use the dataIn and dataOut commareas for binary data exchange between the application and the rule. The SDK (Java) commarea is a byte array formatted as follows:



The total number of bytes in an SDK (Java) commarea is the number of bytes indicated in the header, plus the size of the header. The size of the commarea byte array does not matter as long as it is big enough. If the array is too small, both forms of the call method fail with either a SessionException.INCOMMLENERR (40) or a SessionException.OUTCOMMLENERR (41) error reason code, depending on where the commarea error is detected.

The TIBCO Object Service Broker Execution Environment creates a copy of dataIn and makes the address to the area available to the rule through the APIINHANDLE field of the @SESSION(0) table. For dataOut, the Execution Environment allocates memory for the whole area and copies the area header. The address to dataOut is made available through the APIOUTHANDLE field of the @SESSION(0) table.

Access to dataIn and dataOut using MAP tables is always granted by the system and MAP tables can be used without @MAP registration of the dataIn and dataOut addresses. dataIn is accessible for reading and dataOut for reading and writing. For more information about MAP tables, refer to *TIBCO Service Gateway for Files Installing and Operating*.

When a rule completes successfully, the contents of the dataOut are transferred back from the Execution Environment to the application memory. Consider reducing the number of bytes your rule transmits to the application. You do this by properly reformatting the dataOut header. The Execution Environment, before transmitting data back, reevaluates the dataOut header to determine the correct

number of bytes to send back to the application. Do not use this method to increase the total size of the area. If the contents of the header indicate that the total area size is larger than the original (at the beginning of the rules call), the call fails with a SessionException.DATAOUTCORRUPT (49409) error reason code.

A performance consideration to do with return values, which can be as long as 32 KB:

- In its first form, the call method must create a temporary copy of the return value to perform the code page translation. Therefore this form is not the best option for applications dealing with long return values.
- In its second form, the call method does not convert the rules return value to Unicode. It leaves it up to the application to store and convert the value in the most optimal application-specific manner. No temporary copies of any kind are created during the call in this form.

See Also

- *TIBCO Object Service Broker Programming in Rules* about the rules language and writing rules.
- *TIBCO Object Service Broker Shareable Tools* and *TIBCO Object Service Broker Managing Data* about the @SESSION(0) table and the MAP table.

endMessage

Returns the end message from the last rule called within the session.

```
public String endMessage()
```

Parameters: None.

Return Value: The end message from the last rules call, via a call method, or an error message if the rule failed. If neither an end message nor an error message is available from the rule, returns null. The maximum length of the end message is 148 (Session.MAXENDMSGLEN constant).

Exceptions: None.

Comments endMessage returns a String object, not a null reference, in these cases:

- If the last rules call succeeded, endMessage returns the message generated by the rule via an ENDMSG call, or an empty string if the rule did not generate a message.

- If the last call failed with a SessionException.RULEFAILED (3091) error reason code, endMessage returns a rules error message.

In all other cases, endMessage returns null. The java.lang.Throwable.getMessage method returns a message that is formed according to the above. If the rules call throws an exception with a SessionException.RULEFAILED (3091) error reason code, the message text contains the rules error message instead of a reason code description.

See Also *TIBCO Object Service Broker Shareable Tools* about ENDMSG.

execTran

Equivalent to executing a sequence of startTrans, call, and stopTrans methods.

The changes are committed if the rules call succeeds and rolled back if it throws an exception.

The method supplies the return value from the rule in one of two ways:

- When used in its first form (shown below), execTran returns the return value, via the RETURN rules statement, as a String object.
- In its second form, execTran stores the value, encoded using the SDK (C/C++) /SDK (Java) code page, in a region of a caller-supplied byte array.

```
public String execTran(String transParams,
                       String func,
                       byte[] dataIn,
                       byte[] dataOut) throws SessionException
```

or

```
public int execTran(String transParams,
                       String func,
                       byte[] dataIn,
                       byte[] dataOut,
                       byte[] ruleRetValue,
                       int    ruleRetValueStart,
                       int    ruleRetValueMaxLen) throws
SessionException
```

Parameter:

transParams	The transaction parameter string. Refer to startTrans on page 161.

func	The rules functional expression in a textual form: `RULENAME(ARG1,ARG2,...,ARGN)`.
dataIn	The input commarea: a byte array, formatted in accordance to the commarea format specification (refer to call on page 152).
dataOut	The output commarea: a byte array, formatted in accordance to the commarea format specification.
ruleRetValue	The buffer meant to hold the return value from the rule.
ruleRetValueStart	The index where the program should start storing the return value.
ruleRetValueMaxLen	The maximum number of bytes available in the *ruleRetValue* array.

Return Value: The first form of the method returns the rules return value as a String object.

The second form returns the actual length of the rules return value as returned by the rule. Compare this length to the value passed as *ruleRetValueMaxLen* to determine if truncation took place.

The rules return value is the value that the called rule returns in a RETURN statement.

Exceptions SessionException is thrown if errors are encountered.

The value of *rc* is SessionException.STARTTR_FAILED (10), SessionException.CALLRULE_FAILED (11), or SessionException.STOPTR_FAILED (12), depending on the stage where the first error occurred.

Comments This is a function that operates in several phases:

- Start a transaction
- Call a rule
- Stop the transaction with commit or rollback, depending on the success of the rules call

The only phase able to interrupt the sequence is the startTrans phase. When that phase is successful, the failure of the rules call stores the error information to throw an exception only after the stopTrans method completes. In this case, errors during the stopTrans phase are not reported.

isActive

Returns the activity status of the session.

```
public boolean isActive()
```

Parameters: None.

Return Value: True if the session is active; otherwise, false.

Exceptions: None.

Comments It is possible for a TIBCO Object Service Broker session to become inactive any time after starting (due to network problems, Execution Environment abnormal terminations, and so on). When that happens, the start, stop, startTrans, stopTrans, shutdown, and execTran method calls on this session object fail with an appropriate error reason code. In addition, the session object becomes inactive, so that subsequent calls to these methods fail with a SessionException.CALLOUTOFSEQ (36) error reason code.

Use an isActive method call to determine whether the session is still active. If the session is stopped by a stop method call, by a reset method call, or by a shutdown method call, or if it abends, isActive returns false.

reset

Drops a connection to a session.

```
public void reset
```

Parameters: None.

Return Value: None.

Throws: None.

Comments reset forcefully closes the session by dropping the session connection as opposed to an orderly shutdown by stop. The session does not have to be active for the call to succeed. When a connection is dropped, the Execution Environment generates an error message, and closes the session. All uncommitted data changes are lost.

This method bring the session to an inactive state so that subsequent calls to the start, stop, startTrans, stopTrans, shutdown, and execTran methods fail with a SessionException.CALLOUTOFSEQ (36) error reason code. Subsequent calls to isActive return false.

shutdown

Stops all transactions (committing or rolling back the changes), and stops the session regardless of the errors encountered.

```
public void shutdown(boolean commit) throws SessionException
```

Parameters:

commit	True if the changes made within all transactions need to be committed, otherwise false.

Return Value: None.

Exceptions SessionException is thrown if errors are encountered during the shutdown sequence.

The value of *rc* is either SessionException.STOPTR_FAILED (12) or SessionException.STOPSS_FAILED (9), depending on the stage where the first error occurred.

Comments This method is a sequence of stopTrans method calls and a stop method call. Even if an error occurs, the sequence continues to the end. Information on the first (or only) error is stored and an exception is thrown after the sequence completes, and all the transactions and the session are stopped.

start

Starts an SDK (Java) session.

```
public void start(String sessParam,
                  String codepage) throws SessionException
```

Parameters:

sessParam	The session parameter string. This string must contain CLIHOST and CLIPORT to specify an osMon (Windows or Solaris) or an Execution Environment (z/OS) location. Can contain CLIENDIAN.
codepage	The SDK (C/C++)/SDK (Java) code page to be used for this session. If it is null or empty, the method throws SessionException with an error reason code of SessionException.UNDEFCODEPAGE (161). For valid values, refer to Session on page 150.

Use the session parameters string (*sessParam*) to define various session behavior aspects. There are a number of parameters that are specific for the SDK (Java) (and SDK (C/C++)). These are (names are case insensitive):

CLIHOST	TCP/IP host name of the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS).
CLIPORT	TCP/IP port number of the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS).
CLIENDIAN	Session endian type. This parameter affects the external representation of MAP table fields with internal syntax B and "*" external syntax. Valid entries are: BIG and LITTLE (case insensitive).

CLIHOST and CLIPORT identify the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS) on the network, using the CLIHOST parameter to specify a TCP/IP host name and the CLIPORT parameter to specify a TCP/IP port number.

Make sure that your *sessParam* parameter string contains the CLIHOST and CLIPORT parameters. (CLINODE is not supported by the SDK (Java).) Otherwise, the start method throws an exception with error reason code SessionException.INVNODE (193).

CLIENDIAN provides a way to override the application endian type for a session. This parameter affects the external representation of MAP table fields with numeric internal syntaxes and the "*" external syntax. If CLIENDIAN is not specified, big endian is selected.

Return Value: None.

Throws: This method throws SessionException with rc = SessionException.STARTSS_FAILED (4) if a session cannot be started.

Comments A successful start call changes a session from inactive to active. Use the isActive method to query the current state of a session. If a start call is issued for an object that is already active, the call throws an exception with a SessionException.CALLOUTOFSEQ (36) error reason code. To bring a session back to the inactive state, issue a stop, a reset, or a shutdown method call.

See Also *TIBCO Object Service Broker Parameters* about starting sessions and about session Execution Environment parameters.

startTrans

Starts a transaction.

```
public void startTrans(String transParams) throws SessionException
```

Parameter:

transParams	The transaction parameter string.

Transaction parameters are specified as follows (all characters are case insensitive):

```
BROWSE | UPDATE, TEST | NOTEST, SEARCH=S | I | L, LIBRARY=libname
```

If you omit a parameter, the startTrans method uses the session default value specified in the session parameter string of the start method or of the Session constructor. These session defaults are set by the BROWSE, TEST, SEARCH, and LIBRARY session parameters. For more information, refer to *TIBCO Object Service Broker Parameters*.

Return Value: None.

Throws: This method throws SessionException with rc = SessionException.STARTTR_FAILED (10) if the method fails.

Comments The startTrans method starts a transaction within the session. If the session already has transactions started, the startTrans method starts a child transaction.

If the session is already stopped or abended, the startTrans method throws an exception with an error reason code of SessionException.CALLOUTOFSEQ (36).

Use the stopTrans method to stop the currently active transaction.

If the maximum allowed number of transactions are already running in the session, startTrans throws an exception with an error reason code SessionException.TOOMANYTRANS (106). Refer to *TIBCO Object Service Broker Parameters* for information on the TRANMAXNUM Execution Environment parameter, which sets the maximum value.

stop

Stops an SDK (Java) session.

```
public void stop() throws SessionException
```

Parameters: None.

Return Value: None.

Throws: This method throws SessionException with rc = SessionException.STOPSS_FAILED (9) if the method fails.

Comments If the session is inactive, the stop method call throws an exception with a SessionException.CALLOUTOFSEQ (36) error reason code. You activate a session by using either the Session constructor with the *sessParam* and *codepage* parameters, or a start method call.

If a transaction is still active prior to the stop method call, the call throws an exception with a SessionException.TRANSACTIVE (128) error reason code and the session stays active.

Failures with error reason codes other than SessionException.TRANSACTIVE (128) indicate a failure to complete normal shutdown sequence. The session is rendered inactive anyway.

Use the isActive method to determine the current session state. Use the transNestLevel method to determine the current depth of the transaction stack

stopTrans

Stops the currently active transaction, committing or rolling back the changes.

```
public void stopTrans(boolean commit) throws SessionException
```

Parameters:

commit	True if the changes made within the active transaction need to be committed, otherwise false.

Return Value: None.

Throws: This method throws SessionException with rc = SessionException.STOPTR_FAILED (12) if the method fails.

Comments If the session is not active or no transaction started prior to the call, the call throws an exception with SessionException.CALLOUTOFSEQ (36) error reason code.

The current transaction is destroyed and the session nesting level is decremented even if the call ends with an error. Use the transNestLevel method to determine the current transaction nesting level.

transNestLevel

Returns the transaction nesting level of the session.

```
public int transNestLevel()
```

Parameters: None.

Return Value: The current transaction nesting level of the session. Returns zero if the session is not active or no transaction is started.

Exceptions: None.

userId

Returns the user ID of the session.

```
public String userId()
```

Parameters: None.

Return Value: The session user ID if the session is active, otherwise null.

Exceptions: None.

SessionException Object Methods

SessionException

The SessionException class constructor.

This method has two forms:

```
public SessionException(int rc,
                        int reasonCode)
```

or

```
public SessionException(String ruleErrorText)
```

In the first form, SessionException constructs a SessionException object with the specified error code and error reason code. The resulting exception error message string (available using a java.lang.Throwable.getMessage call) contains the following: {error code description} - {error reason code description available via errorReasonDescr}. For example, if a startTrans method throws an exception due to the interface calls being out of sequence, the error message string is: "JCLI transaction startup failed - Interface calls are out of sequence". The SDK (Java) uses this form of the SessionException constructor to report all errors with an error reason code other than SessionException.RULEFAILED (3091).

In the second form, SessionException constructs a SessionException object with the SessionException.CALLRULE_FAILED (11) return code, a SessionException.RULEFAILED (3091) error reason code, and a specified error message. The resulting exception error message string (available using a java.lang.Throwable.getMessage call) contains the following: {error code description} - {ruleErrorText}. For example, if a call("MYRULE"...) throws an exception because the MYRULE rule encountered an access error on table named MYTABLE, the error message string is "JCLI rule call failed - Access error on TABLE "MYTABLE"". The error reason code for these errors is always SessionException.RULEFAILED (3091). The SDK (Java) uses this form of the SessionException constructor to report rules failures, passing the string available via endMessage at the end of the call as a *ruleErrorText* parameter.

Parameters:

rc	The error code.
reasonCode	The error reason code.

ruleErrorText	The rules error message.

Return Value: None.

Comments You can access the exception error message string using the getMessage() method or the toString() method defined in the java.lang.Throwable class.

errorReasonDescr

Returns a textual description of a particular SDK (Java) error reason code.

```
public String errorReasonDescr(int reasonCode)
```

Parameters:

reasonCode	The error reason code.

Return Value: For a list of error reason codes, refer to Appendix A, SDK (C/C++) and SDK (Java) Error Reason Codes, on page 179.

Comments Use the reasonCode method to retrieve the value of a particular error reason code.

reasonCode

Returns an error reason code for an SDK (Java) error.

```
public int reasonCode()
```

Parameters: None.

Return Value: For a list of error reason codes, refer to Appendix A, SDK (C/C++) and SDK (Java) Error Reason Codes, on page 179.

Comments Use the errorReasonDescr method to retrieve a textual description of a particular error reason code.

rc

Returns an SDK (Java) operation error code.

```
public int rc()
```

Parameter: None.

Return Value: Depending on the method that failed, the error returned is one of the following:

- SessionException.STARTSS_FAILED= 4
- SessionException.STOPSS_FAILED= 9
- SessionException.STARTTR_FAILED= 10
- SessionException.CALLRULE_FAILED= 11
- SessionException.STOPTR_FAILED= 12

Misc Object Methods

commCreate

Creates a new byte array and formats it according to the commarea specification.

The size of the new byte array is calculated based on the supplied segment structure. In the second form, the structure is assumed to have one segment of segmentSize bytes

```
public static byte[] commCreate(int[] segmentSizes)
```

or

```
public static byte[] commCreate(int segmentSize)
```

Parameters:

segmentSizes	Array of segment sizes.
segmentSize	Size of the only commarea segment.

Return Value: A new commarea byte array.

Exceptions First form: OutOfMemoryError, NullPointerException.

Second form: OutOfMemoryError.

Comments The part of the array that belongs to the segment bodies is not initialized.

commFormat

Formats a byte array according to the commarea specification.

In the second form, the commarea is assumed to contain one segment of *segmentSize* bytes.

```
public static void commFormat(byte[] area,
                              int[]  segmentSizes)
```

or

```
public static void commFormat(byte[] area,
                              int    segmentSize)
```

Parameters:

area	Array to format.
segmentSizes	Array of segment sizes.
segmentSize	Size of the only commarea segment.

Return Value: None.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

commSegmentInd

Returns the offset of a specified segment in a commarea.

```
public static int commSegmentInd(byte[] area,
                                 int    segmentNum)
```

Parameters:

area	Byte array formatted according to the commarea specification.
segmentNum	The segment number.

Return Value: The offset of the beginning of the segment body, or 0 if the segment does not exist.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

commSegments

Returns the number of segments in a commarea according to its header.

```
public int commSegments(byte[] area)
```

Parameters:

area	Byte array formatted according to the commarea specification.

Return Value: The number of segments in a commarea according to its header.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

commSegSize

Returns the size of a given commarea segment.

```
public int commSegSize(byte[] area,
                       int     segmentNum)
```

Parameters:

area	Byte array formatted according to the commarea specification.
segmentNum	The segment number.

Return Value: The segment size, or 0 (zero) if the segment does not exist.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

commSize

Calculates the number of bytes in a commarea according to its header.

```
public int commSize(byte[] area)
```

Parameters:

area	Byte array formatted according to the commarea specification.

Return Value: The total size of a commarea according to its header.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

commSizeCalc

Calculates the number of bytes needed for a commarea with the specified structure.

This method has two forms:

```
public static int commSizeCalc(int[] segmentSizes)
```

or

```
public int commSizeCalc(int segmentSize)
```

In the first form, the commarea has a structure supplied by the input array.

In the second form, the commarea has one segment of a given size.

Parameters:

segmentSizes	Array of segment sizes.
segmentSize	Size of the only commarea segment.

Return Value: The size of an area with the supplied structure.

Exceptions For the first form, since no verification of input parameters is performed, standard array access exceptions can be thrown. The second form throws no exceptions.

readInt

Reads 4 bytes from a byte array and returns a value of type int according to what these bytes represent in big-endian format.

```
public static int readInt(byte[] b,
                          int    offset)
```

Parameters:

b	Input byte array.
offset	Starting index.

Return Value: The value of type int that the specified 4 bytes represent in big-endian format.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

readShort

Reads 2 bytes from a byte array and returns a value of type short according to what these bytes represent in big-endian format.

```
public static short readShort(byte[] b,
                              int    offset)
```

Parameters:

b	Input byte array.
offset	Starting index.

Return Value: The value of type short that the specified 2 bytes represent in big-endian format.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

writeInt

Writes, into a byte array, 4 bytes of a big-endian byte representation of a value of type int.

```
public static void writeInt(int    n,
                            byte[] b,
                            int    offset)
```

Parameters:

n	Input value of type int.
b	Output byte array.
offset	Starting index.

Return Value: None.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

writeShort

Writes, into a byte array, 2 bytes of a big-endian byte representation of a value of type short.

```
public static void writeShort(short  s,
                              byte[] b,
                              int    offset)
```

Parameters:

s	Input value of type short.
b	Output byte array.

`offset`	Starting index.

Return Value: None.

Exceptions Since no verification of input parameters is performed, standard array access exceptions can be thrown.

Sample Application Using the SDK (Java)

Compiling and Running the Sample Program

The sample is available in the /install_path/JavaCLI/src folder.

Windows

Using the Sun JDK 1.3, the sample program was compiled and executed with the following assumptions:

- The Java program resides in the current directory, as JCLISAMP.java
- The OS_ROOT environment variable is initialized to the TIBCO Object Service Broker install path
- The .jar file for the SDK (Java) is install_path\bin\cli.jar
- The TIBCO Object Service Broker Data Object Broker and TIBCO Object Service Broker monitor (osMon) are running, with the osMon listening on port 9068

Compile and execute the program using these commands:

1. SET CLASSPATH=.;%OS_ROOT%\bin\cli.jar;%CLASSPATH%
2. javac JCLISAMP.java
3. java JCLISAMP clihost=localhost,cliport=9068 ISO8859-1

Solaris

The sample program was compiled and executed with the following assumptions:

- The SDK (Java) .jar file exists as install_path/bin/cli.jar
- The source file JCLISAMP.java exists in the current directory
- The OS_ROOT environment variable is initialized to the TIBCO Object Service Broker install path
- The TIBCO Object Service Broker Data Object Broker and TIBCO Object Service Broker monitor (osMon) are running, with the osMon listening on port 9068

Use the following commands to compile and run the sample program:

1. export CLASSPATH=.:$OS_ROOT/bin/cli.jar:$CLASSPATH

2. javac JCLISAMP.java
3. java JCLISAMP clihost=localhost,cliport=9068 ISO8859-1

The sample uses the HURON1 TIBCO Object Service Broker user ID with a password of HURON1.

Sample Rule Called by Program

This is a sample of a rule that creates an occurrence of a TDS table, generates an end message, and returns a value. On completion of the rule, the changes are not committed because the transaction is still active. The SDK (Java) program explicitly stops the transaction by issuing stopTrans with a true flag or a false flag to indicate whether the changes are to be committed.

```
RULE EDITOR ===>                                                  SCROLL: P
 TC007124RU02;
 _
 _ ------------------------------------------------------------------------------
 _ ------------------------------------------------------------+---------------
 _ TC007124TA01.TEXT = 'RULE "TC007124RU02" IS CALLED';         | 1
 _ INSERT TC007124TA01;                                         | 2
 _ CALL ENDMSG('END MESSAGE GENERATED BY RULE "TC007124RU02"'   | 3
 _   );                                                         |
 _ RETURN('RETURN VALUE OF RULE "TC007124RU02"');               | 4
 _ ------------------------------------------------------------------------------
 _
```

Sample Table Referenced by a Rule

The table is defined as follows:

```
COMMAND==>                          TABLE DEFINITION

      Table: TC007124TA01     Type: TDS   Unit: TC007124           IDgen: Y

   Parameter Name  Typ Syn Len Dec Class             '      Event Rule   Typ Acc
  ----------------  -   -  --- --    -              '   ---------------- -   -
_ LOCATION           I   C   16  0    L              ' _
_                                                   ' _
     Field Name   Typ Syn Len Dec Key Ord Rqd    Default         Reference
  ---------------- -   -  ---- --  -   -   - ---------------- ----------------
_ KEY              I   B     4  0  P
_ TEXT             S   C    50  0
_
_
_
_
_
_
_
_
_
_
PFKEYS: 3=END 12=CANCEL 22=DELETE 13=PRINT 14=FIELDS 21=DATA 2=DOC
New table definition
```

Output from Program

The output from the program is as follows:

```
Start session completed
Start transaction completed
Rule call TC007124RU02 completed; Return value='RETURN VALUE OF
RULE "TC007124RU02"'; End message='END MESSAGE GENERATED BY RULE
"TC007124RU02"'
Stop transaction completed
Stop session completed
```

When the rule is done, a row is added to the table:

```
EDITING TABLE   :   TC007124TA01
COMMAND ==>
                                                                    SCROLL: P
      KEY                              TEXT
_ ----------- ------------------------------------------------------
_           1 RULE "TC007124RU02" IS CALLED
_
_
_
_
_
_
_
_
_
_


PFKEYS: 4=INS 16=DEL 5=FIND NXT 6=CHG NXT 18=EXCL 19=SHOW 3=SAVE 12=CANCEL
```

Appendix A SDK (C/C++) and SDK (Java) Error Reason Codes

This appendix lists the error reason codes for the C and C++ SDK and the Java SDK.

Topics

Listing of the Reason Codes

Code Values and Explanations

The following table lists the error reason codes. A listing of symbols is available for your use in install_path/cli/include/oscl.

Reason Code	Symbolic Name – for full C name, add "CLI_"; for full Java name, add "SessionException.")	Explanation
36	CALLOUTOFSEQ	Interface calls are out of sequence.
37	NOSTANDBYSESS	No standby sessions active in the Execution Environment.
38	INCOMMAERROR	Input commarea storage is inaccessible.
39	OUTCOMMAERROR	Output commarea storage is inaccessible.
40	INCOMMLENERR	Input commarea length error.
41	OUTCOMMLENERR	Output commarea length error.
43	DISPCODEPAGEV	DISPLAYCODEPAGE value not supported.
53	CICSEENOTSUPP	This version of HRNDRAPI does not support a CICS Execution Environment.
65	INVALIDSSPARM	Invalid session parameters.
66	INVALIDUSERID	TIBCO Object Service Broker user ID is longer than eight characters.
67	INVALIDCHARSET	Invalid character set name.
68	INVALIDEXECMO	Invalid execution mode.
69	SECLOGONFAIL	Security login fail.
70	SECLOGOFFFAIL	Security logout fail.

Reason Code	Symbolic Name – for full C name, add "CLI_"; for full Java name, add "SessionException.")	Explanation
71	SESSSTORINITF	Session scope storage initialization fail.
72	SESSSTORTERMF	Session scope storage termination fail.
73	USEREXITRC	Session rejected by user exit.
74	UNKNOWNEXITRC	Unknown return code from user exit.
75	NLSINITFAIL	NLS initialization failed.
86	INVALIDTRANOP	Invalid transaction option.
96	NORULENAME	Rule name not supplied.
97	RULENAMELNERR	Rule name length error.
99	INVALIDENDTR	Invalid stop transaction parameter.
102	COMMITFAIL	Commit failed.
103	ROLLBACKFAIL	Rollback failed.
104	RULEARGERROR	Rule name or argument syntax error.
106	TOOMANYTRANS	Too many transactions in a session.
128	TRANSACTIVE	Transaction still active.
160	INCOMPATVERSN	Incompatible client/server version.
161	UNSUPPCODEPAG	Unsupported code page specified.
163	SESSINITERROR	Session control storage not available.
164	CICSTASKFAIL	Start CICS task failed.
165	ENVINITERROR	Environment initialization error.
166	MSGTOOLONG	SDK (C/C++) message length exceeds maximum.

Reason Code	Symbolic Name – for full C name, add "CLI_"; for full Java name, add "SessionException.")	Explanation
167	MSGSTORNA	SDK (C/C++) message storage not available.
168	DATAOUTTOOLONG	dataOut length exceeds maximum.
169	DATAOUTSTORNA	dataOut storage not available.
170	USIDNOTSUPPL	User ID not supplied.
171	LOGONINVALID1	User ID or password in not valid. [a.]
171	USIDINVALID	User ID or password in not valid. [a.]
172	USIDSUSPENDED	User ID suspended.
173	USIDCANTACCES	User ID cannot access TIBCO Object Service Broker at this time.
174	PSWDNOTSUPPL	Password not supplied.
175	LOGONINVALID2	User ID or password in not valid. [a.]
175	PSWINVALID	User ID or password in not valid. [a.]
176	PSWDEXPIRED	Password expired, new password missing.
177	NEWPSWDINVAL	New password not valid.
178	PSWDUPGRADEFA	Password upgrade fail.
179	PSWDDECRYPTFA	Password decrypt fail.
180	CORRDATA	Corrupt message received from server.
193	INVNODE	Invalid host/node or host/port specification.
194	UNDEFNODE	Undefined node.
195	COMMFAILURE	Communication failure.

Reason Code	Symbolic Name – for full C name, add "CLI_"; for full Java name, add "SessionException.")	Explanation
196	INVDATA	Invalid message received from server.
197	MEMORY	Client cannot allocate memory for operation.
198	BADCOMMFORMAT	Unsupported commarea format.
199	SESSINVALID	Invalid CLI_SESSION parameter.
200	SESSCANCELLED	Session was canceled or terminated.
201	BUFTOOSMALL	Buffer for rules return value is too small (< 3 bytes).
208	SINGLEUSER	This version of the SDK (C/C++) client allows connections to host "localhost" only.
3088	SESSPARMTOOLONG	Session parameter string is too long (> 65535).
3089	NODENOTSUPPORTED	CLINODE parameter is not supported by the SDK (Java).
3090	RULEEXPRTOOLONG	Rules expression is too long.
3091	RULEFAILED	Rules call failed.
3092	UNDEFCODEPAGE	Undefined SDK (C/C++)/SDK (Java) code page supplied.
3093	INVRETVALIND	Invalid rules return value start index or maximum length.
49408	UNIDENTEEERROR	Internal Execution Environment error or unidentified Execution Environment error.
49409	DATAOUTCORRUPT	Output commarea is corrupted after rules call.

a. Note that logon failures, depending on particular TIBCO Object Service Broker system can be reported by either LOGONINVALID1 or LOGONINVALID2 reason codes. Both codes mean that a user cannot login using this User ID/password combination. USIDINVALID and PSWDINVALID reason codes have the same values as LOGONINVALID1 and LOGONINVALID1 and are present for compatibility reasons only.

Index

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

W